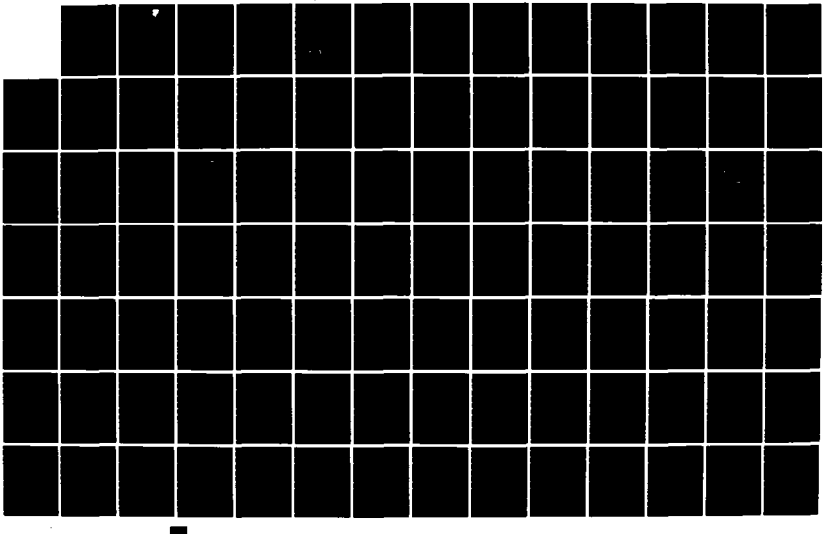
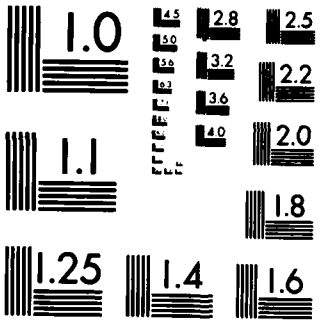
AD-A147 289 SOFTWARE TEST HANDBOOK: SOFTWARE TEST GUIDEBOOK VOLUME 2(U) BOEING AEROSPACE CO SEATTLE WA ENGINEERING TECHNOLOGY DIV E PRESSON MAR 84 RADC-TR-84-53-VOL-2 1/3

UNCLASSIFIED F30602-82-C-0059 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

12

RADC-TR-84-53, Vol II (of two)
Final Technical Report
March 1984

SOFTWARE TEST HANDBOOK Software Test Guidebook

Boeing Aerospace Company

Edward Presson

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-84-53, Vol II (of two) has been reviewed and is approved for publication.

APPROVED: Frank S LaMonica

FRANK S. LaMONICA
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Acting Technical Director
Command & Control Division

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-84-53, Vol II (of two)
6a. NAME OF PERFORMING ORGANIZATION Boeing Aerospace Company Engineering Technology 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COEE)
6c. ADDRESS (City, State and ZIP Code) Seattle WA 98124	7b. ADDRESS (City, State and ZIP Code) Griffiss AFB NY 13441
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COEE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-82-C-0059
8c. ADDRESS (City, State and ZIP Code) Griffiss AFB NY 13441	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. 63728F; PROJECT NO. 2527; TASK NO. 02; WORK UNIT NO. 09
11. TITLE (Include Security Classification) SOFTWARE TEST HANDBOOK Software Test Guidebook	
12. PERSONAL AUTHOR(S) Edward Presson	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM Mar 82 TO Sep 83	14. DATE OF REPORT (Yr., Mo., Day) March 1984 15. PAGE COUNT 288
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Testing Software Computer Programs
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of the Software Test Handbook effort was to provide Air Force software developers with guidelines and methodology for the effective use of higher order language (HOL) software testing techniques and in the selection of automated tools for the testing of computer programs.

The effort resulted in a two volume final technical report. Volume I, the Final Technical Report, describes the total contractual effort. This report, Volume II - Software Test Guidebook, contains the guidelines and methodology resulting from the effort. In addition, it contains the following: (1) summary descriptions of the testing techniques, (2) an extensive bibliography, (3) typical paragraphs that can be included in software development Statements of Work (SOW) to specify the use of advanced software testing techniques by the contractor during the testing and verification phases of a contracted software development, and (4) a cross-reference to government and commerically available catalogs listing automated test tools that support the various testing techniques.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Frank S. LaMonica	22b. TELEPHONE NUMBER (Include Area Code) (315) 330-3977 22c. OFFICE SYMBOL RADC (COEE)

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

PREFACE

The Software Test Guidebook provides guidelines for the selection of state-of-the-art HOL software testing techniques appropriate to various software development environments and various types of software. The guidelines cover the DT&E, OT&E, and V&V phases, and may be used as an adjunct during maintenance phases. These guidelines are incorporated in a table-driven format that define increasingly thorough levels of testing based on a testing "level of confidence."

The guidebook provides summary descriptions of all the testing techniques, and an extensive bibliography. It includes typical paragraphs than can be included in Air Force software development statements-of-work to specify the use of advanced software testing techniques by the contractor during the testing and verification phases of a contracted development. A cross reference to catalogs listing automated tools provide a method for locating software tools that support the selected testing approaches.

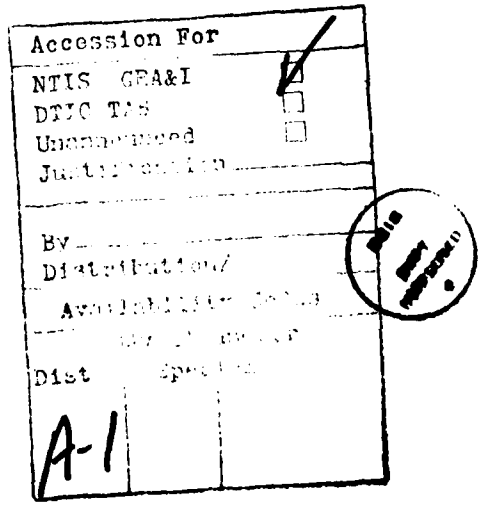


TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

APPENDICES

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AD	Armament Division
ADC	Aerospace Defense Center
ADP	automatic data processing
AFTI	advanced fighter technology integration
ALCM	air-launched cruise missile
ASD	Aeronautical Systems Division
ATE	automatic test equipment
ATO	Air Tasking Order
BITS	built-in test
C^3	command, control, and communications
CAFMS	Computer Assisted Air Force Management System
CCPDS	Command Center Processing and Display System
CINCSAC	Commander-in-Chief Strategic Air Command
CITS	central integrated test systems
COM	computer output microfiche
COSMIC	(Computer Software Management and Software Organization)
CPC	computer program component
CPCI	computer program configuration item
CPU	central processor unit
CRISP	computer resources integrated support plan
CSC	computer software component
CSCI	computer software configuration item
CSOC	Consolidated Space Operations Center
DACS	Data and Analysis Center for Software
DGZ	designated ground zero
DID	data item description
DT&E	development test and evaluation
DT&V	development test and verification
EDP	electronic data processing

ELINT	electronic intelligence
FCA	functional configuration audit
FOT&E	follow-on operational test and evaluation
FSEC	Federal Software Exchange Center
HDM	hierarchical design methodology
HOL	higher order language
ICBM	intercontinental ballistic missile
ICP	International Computer Programs, Inc.
IDHS	intelligence data handling system
IOT&E	initial operational test and evaluation
IUS	inertial upper stage
IV&V	independent verification and validation
JINTACCS	Joint Interoperable Tactical Air Command and Control System
JSCS	Joint Strategic Connectivity Staff
JSTPS	Joint Strategic Target Planning Staff
LRU	line replaceable unit
MATE	Modular Automatic Test Equipment (program)
NBS	National Bureau of Standards
NMCC	National Military Command Center
O&S	operations and support
OFP	operational flight programs
OPR	Office of Primary Responsibility
OT&E	operational test and evaluation
PCA	physical configuration audit
PDR	preliminary design review
PROM	programmable read-only memory

QOT&E	qualification operational test and evaluation
RADC	Rome Air Development Center
RCI	Reifer Consultants, Inc.
RFP	request for proposal
SAC	Strategic Air Command
SCF	Satellite Control Facility
SD	Space Division
SDL	software development laboratory
SILTF	System Integration Laboratory and Test Facility
SIOP	Single Integrated Operational Plan
SLBM	submarine-launched ballistic missile
SOLARS	SAC On-Line Analysis and Retrieval System
SOW	statement of work
SPO	System Program Office
SRA	Software Research Associates
SREM/REVS	Software Requirements Engineering Methodology/Requirement Engineering Validation System
SRU	shop replaceable unit
T&E	test and evaluation
TAC	Tactical Air Command
TACC	Tactical Air Control Center
TACS	Tactical Air Control System
TCL	testing confidence level
TRD	Test Requirements Document
TRICOMS	Triad Computer System
TTY	teletypewriter
USAF	United States Air Force
UUT	unit under test
V&V	verification and validation
WWMCCS	Worldwide Military Command and Control System

1.0 SOFTWARE TEST GUIDEBOOK

1.1 INTRODUCTION

In the past decade many specialized software testing philosophies and testing techniques have evolved. Many of these techniques have been implemented as computer programs; that is, they have been automated. Automated testing tools can enhance the effectiveness of a software test phase.

The Software Test Guidebook is designed to assist Air Force software developers and maintainers in the effective use of higher order language (HOL) software testing techniques and in the selection of automated tools for the testing of computer programs. Using this guidebook, the user can select the appropriate state-of-the-art testing techniques for specific software and determine the availability of automated testing tools that implement the selected techniques. Guidelines and methodologies are specified for understanding and applying automated state-of-the-art testing techniques in various types of Air Force software development and support environments.

Guidelines provided in the guidebook can be applied during the computer software coding and checkout, test and integration, and operation and support phases of development test and evaluation (DT&E), operational test and evaluation (OT&E), and verification and validation (V&V), as defined in AFR 80-14, "Research and Development Test and Evaluation," and AFR 800-14, Volume II, "Acquisition and Support Procedures for Computer Resources in Systems". The guidebook may also be useful in maintenance and modification environments as an adjunct to test planning.

The method by which the user selects testing techniques is based on the use of rating tables found in section 2.0. These tables permit a compact representation of many considerations. A discussion of these considerations is found in section 1.4.

The evaluation of testing techniques is based on a survey of current technical literature, such as journals, conference proceedings, and textbooks. The evaluation is also based on surveys of software engineers who are experienced in software testing and in the use of modern testing techniques and automated test tools. The testing technique recommendations are based on quantitative and qualitative information and should be regarded as guidelines, not as rigid rules.

1.2 OUTLINE OF THE GUIDEBOOK

The Software Test Guidebook comprises six major sections and five appendices, as shown in figure 1.2-1. A brief summary of its contents is found in the following paragraphs.

Section 1.0 states the objectives of the guidebook, describes its outline and content, and discusses its applications.

Section 2.0 presents a compact set of instructions, guidelines, and tables for selecting software testing techniques. It also includes sections on the selection of software support tools and on test completion criteria.

Section 3.0 contains a list of the major catalogs that provide information on automated software tools. It has an index for determining the availability of existing software tools that support the techniques selected by the guidebook user.

Section 4.0 defines the terms used in the taxonomy of testing techniques and gives a detailed description of state-of-the-art testing techniques. These descriptions discuss the technique and related considerations such as cost, user training, and hardware requirements.

Section 5.0 discusses the software life cycle, software acquisition cycle, and the normal phases of testing, as defined in AFR 80-14 and 800-14. This section is a supplement to the guidebook and does not replace the Air Force regulations.

Section 6.0 has several model statement of work (SOW) paragraphs that may be used as prototypes by Air Force acquisition managers in preparing a request for proposal (RFP).

The appendices describe five Air Force mission areas: armament; avionics; command, control, and communication (C^3); missile/space; and mission/force management. Each appendix lists software functions characteristic of the computer programs developed within that mission. Each function is classified according to the software categories used in section 2.0.

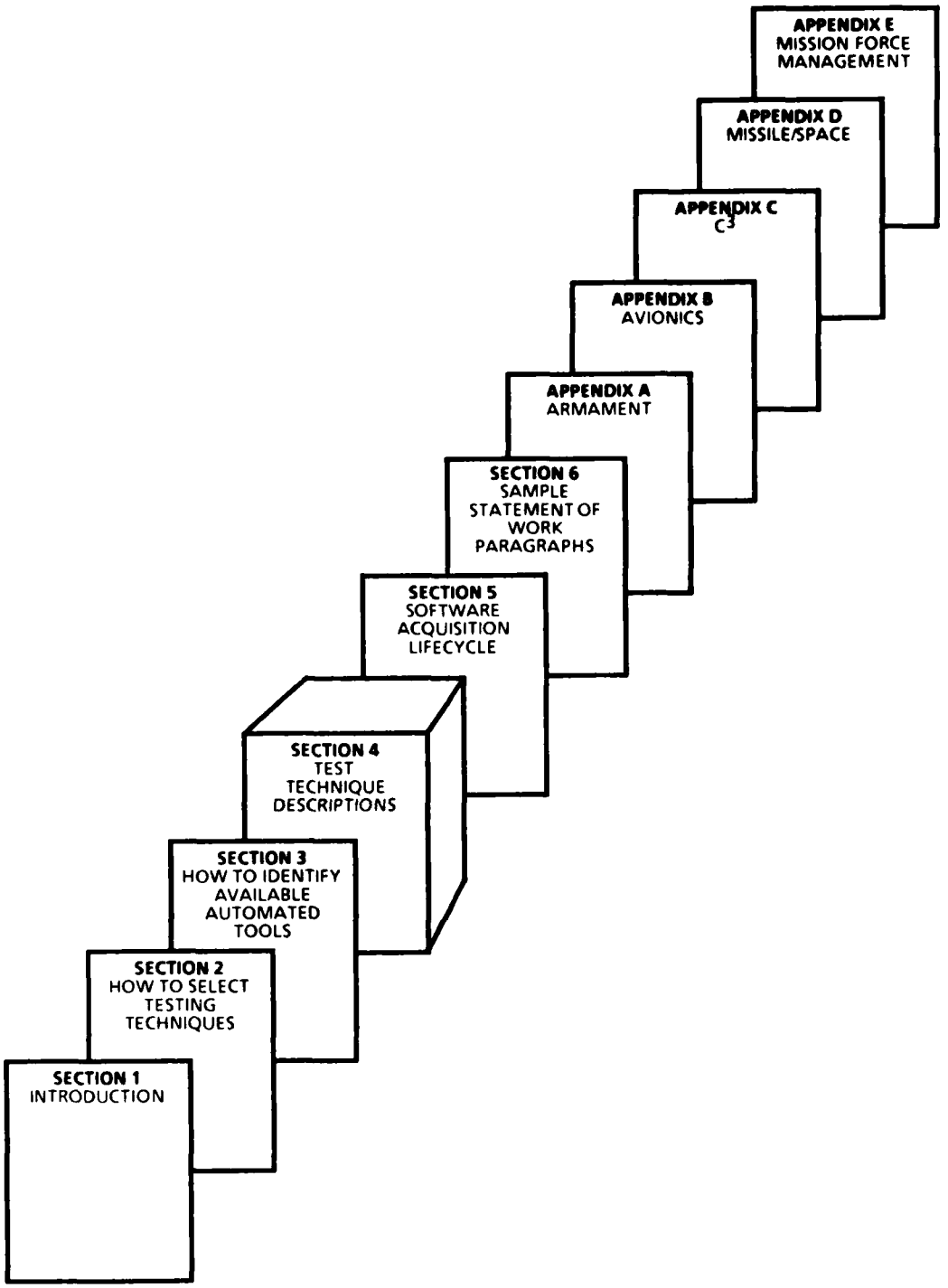


Figure 1.2-1. Guidebook Organization

1.3 APPLICATIONS OF THE GUIDEBOOK

The principal purpose of this guidebook is to select state-of-the-art testing techniques for organic software testing. In addition, the guidebook can be used for preparation of a Statement of Work and for evaluation of proposals.

All three applications of the guidebook use the tables and guidelines of section 2.0 to determine appropriate software testing techniques.

1.3.1 Organic Software Testing

The guidebook can be used as an aid in planning organic software testing and can recommend state-of-the-art software testing techniques based on any of three different kinds of information:

a. The type of software being tested.
b. The test phase or test objective.
c. Knowledge of specific kinds of software errors to be tested.

The guidebook has five appendices to categorize software and environment characteristics of five Air Force mission areas. The guidebook will provide a candidate list of state-of-the-art testing techniques that can be applied to the specific software to be tested.

1.3.2 Preparation of a Statement of Work

The guidebook can be used in preparing SOW paragraphs for use in an RFP. As described in section 1.3.1, the guidebook can be used to identify applicable state-of-the-art testing techniques. The guidebook also provides sample SOW paragraphs in section 6.0 for defining levels of testing at various degrees of formal contractual control. An Air Force acquisition manager can use the guidebook to select the types and levels of testing and to prepare SOW paragraphs with corresponding requirements.

1.3.3 Evaluation of Proposals

The guidebook can be used by an Air Force software acquisition manager to evaluate proposals. The acquisition manager can consider the type of software and its mission,

criticality, and environment and use the guidebook to determine recommended state-of-the-art software testing techniques. The proposal test plan can then be evaluated by comparing it to the guidebook recommendations, providing an additional benchmark in the proposal evaluation.

1.4 CONSIDERATIONS USED IN SOFTWARE TESTING TECHNIQUES SELECTION TABLES

Software testing techniques are selected using the tables in section 2.0. So learning the selection process is a matter of learning to use the tables.

A methodology based on the use of tables was adopted because it simplifies the selection process and condenses a large amount of information. The tables can be updated easily as state-of-the-art software testing continues to evolve.

Each table entry represents many considerations. The tables were constructed based on state-of-the-art software testing theory and a survey of current Air Force mission application test requirements. The following considerations were used in rating and evaluating the testing techniques.

1.4.1 Error Detection and Location Capability

This is the relative success at detection of specific error types. It is also the precision with which the technique locates the software error so that it can be understood, analyzed, and corrected efficiently.

1.4.2 Side Effects and Benefits

These are additional benefits of the technique. For example, automation of some techniques may provide output that can be used in the product documentation of the software being tested.

1.4.3 Cost and Schedule Impact

A testing technique may have several significant cost impacts. The first is its development or acquisition cost; the second is its cost of application. The application

cost may result from computer resources required by the technique or by special skills or training required (discussed separately in a following paragraph). The ultimate cost benefit will be in the savings derived from a reliable software product during the operational phase. That is, both costly failures and the cost of correcting undetected errors will be reduced. Side benefits, such as output that can be used as part of the documentation, have a positive schedule benefit.

1.4.4 Management Impact

All techniques were evaluated for their benefit to management. Some approaches provide visibility on the progress of the software development project. If a technique provides additional visibility to management, it was rated more productive. It should be noted that some techniques, such as peer code review, specifically exclude management visibility in order to achieve their goal.

1.4.5 Training

Some techniques are very difficult to understand and may require extensive training of personnel, using personnel with special skills (algebraic and symbolic analysis). Other techniques use common skills that are simple to apply (e.g., design and code walk-throughs).

1.4.6 Usage Constraints

General usage constraints were considered in evaluating the techniques. Generic techniques were evaluated; no specific automated tools were included. At present, an automated tool incorporating a technique may be available only on the computers of a specific vendor, which would considerably limit availability and applicability of that tool. In the future, however, tools incorporating the technique may be available on many machines. The tables in the guidebook rate generic software testing techniques, not specific software test tools.

1.4.7 Typical Computer Resources

Each technique was evaluated in terms of its computer resource (time and storage) requirements compared to the potential benefits. Some techniques such as peer code

review require little or no computer resources; whereas, other techniques such as random or real-time testing may use substantial amounts of central processor unit (CPU) time, primary and secondary storage, or a considerable number of other computer resources.

1.4.8 Level of Human Interaction

The techniques included in the tables vary in the required level of human interaction. Human interaction must be considered at two levels: (1) a comparison of the amount of test engineer time required versus the potential benefits and (2) the degree of expertise needed to effectively use the technique. The greater the relative amount of time or level of expertise, the lower the technique rating.

1.4.9 Usefulness of Technique in Supporting Modern Programming Practices

If the technique encourages modern programming practices used by the software developers the technique was given a relatively high rating.

2.0 SELECTION OF SOFTWARE TESTING TECHNIQUES

2.1 THREE PATHS OF TECHNIQUE SELECTION

This section provides step-by-step instructions for selecting software testing techniques appropriate to software products and conditions, such as type of software or development phase. In addition, this section discusses important considerations in selecting testing techniques. Extended discussions of the techniques are provided in section 4.0.

The guidebook methodology provides three paths, shown in figure 2-1, that can be followed to determine appropriate software testing techniques. If possible, all three paths should be used to ensure that no relevant software techniques are omitted from consideration.

The three paths offered for the selection of testing techniques should not be confused with the three major applications of the guidebook described in section 1.3. All three paths are appropriate for each application.

In the first path (sec. 2.2) the selection of software testing techniques is based principally on the category of software being tested. For example, a real-time executive would require different testing approaches than those used for a postmission data analysis computer program.

The second path (sec. 2.3) uses information from the test phase and test objectives to select testing techniques. For example, an extensive real-time test would be appropriate after the real-time executive software had been successfully integrated and preliminary testing completed. The real-time test normally would not be appropriate during the module testing phase. The software acquisition lifecycle and associated test phases are described in section 5.0.

The third path (sec. 2.4) selects testing techniques based on the knowledge of software error categories that are currently occurring or have previously occurred on similar software development projects.

Use as many of the three paths as you have information to support, because each path

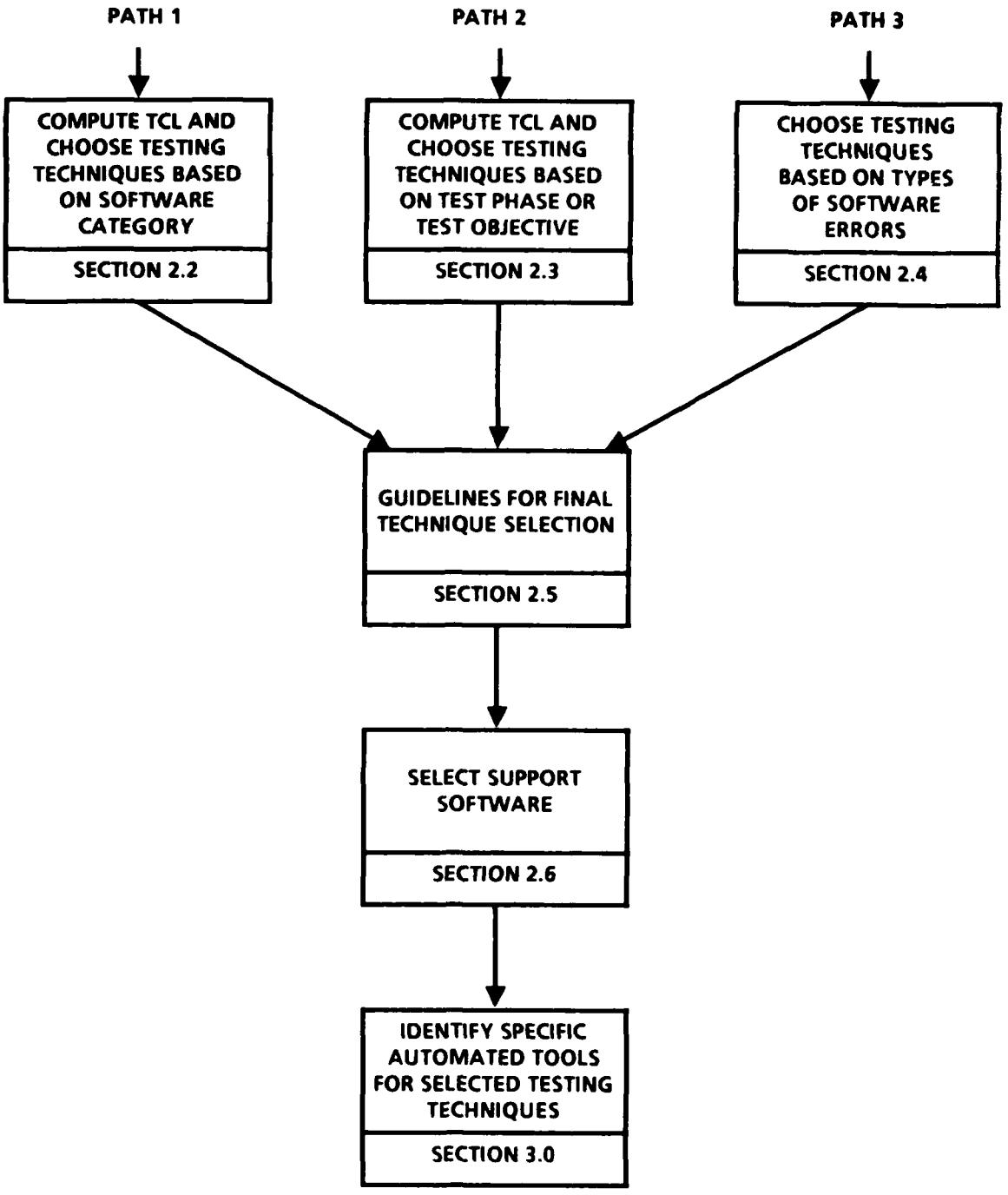


Figure 2-1. Three Paths for Selecting Software Testing Techniques

introduces unique considerations. No technique used alone has proved effective in detecting and isolating all software errors, so no one technique or methodology is a cure-all.

Section 2.5 presents guidelines for final selection of testing techniques from the candidate list built using paths 1, 2, and 3. Section 2.6 contains a table to aid in selecting support tools appropriate to the selected techniques. Section 2.7 discusses considerations concerning when to stop testing, and section 2.8 has an example of the application of the three paths used in this guidebook to select testing techniques.

The three paths use tables to select software testing techniques. There are five basic tables. Figure 2-3 and the TCL worksheet allow the reader to compute the testing confidence level (TCL) appropriate for the software to be tested. This TCL is used in two subsequent tables. Figure 2-7 lists 18 software categories used in this guidebook. Figure 2-8 uses the TCL and the category of software to be tested in order to select testing techniques. Figure 2-11 uses the TCL and the test phase or test objectives to accomplish the same goal. Figure 2-14 rates software testing techniques for their ability to detect various categories of software errors.

Additional information regarding the benefits and disadvantages of a candidate technique can be found in section 4.0. Consider how each candidate testing technique will impact other techniques and how they reinforce or conflict with each other.

Use of this guidebook will provide a set of candidate software testing techniques applicable to the software testing problem at hand. However, simply because the guidebook recommends a technique does not always mean that it is feasible or cost effective in your environment. Judgment must be used in selecting techniques that are both applicable and feasible. The expanded technique descriptions provided in section 4.0 will aid in this determination.

2.2 PATH 1–SOFTWARE CATEGORY

An outline of this path is shown in figure 2-2. The individual steps are discussed in detail in the following paragraphs.

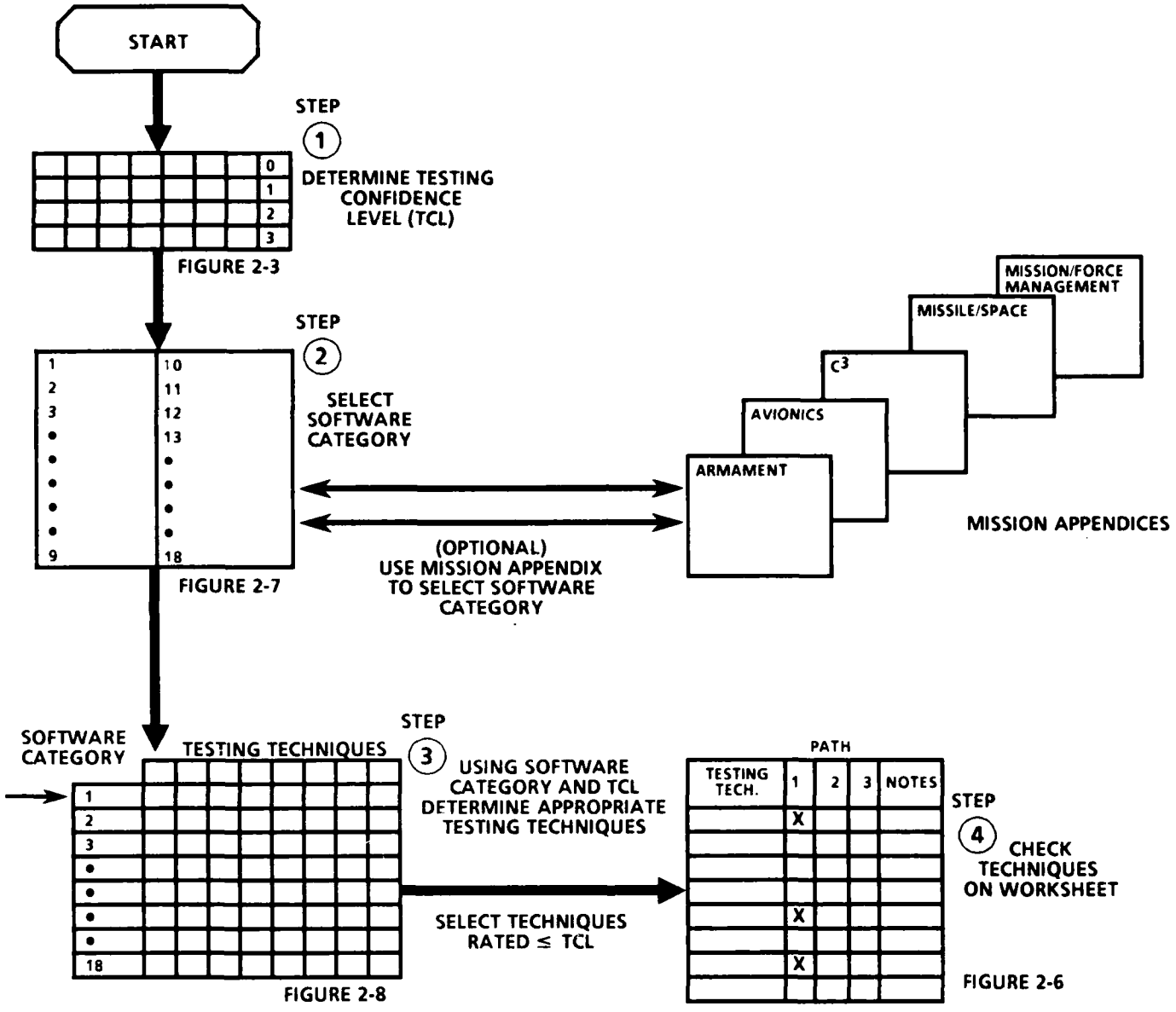


Figure 2-2. Path 1–Selection of Software Testing Techniques by Software Category

2.2.1 Step 1

The first step is to choose a testing confidence level appropriate for the software product you are considering. The TCL is determined by using figure 2-3. The worksheet illustrating the TCL computation is shown in figure 2-5. Section 2.9 contains masters of blank worksheets.

Different TCL's may be assigned to an individual module, as well as to a complete software entity such as a computer program component (CPC) or computer program configuration item (CPCI). (The proposed MIL-STD-SDS uses the new terms, computer software component (CSC) and computer software configuration item (CSCI), for CPC and CPCI.) An example of such usage would be a software system containing a secure operating system, which requires a very high TCL, and ordinary software controlled by that operating system, where a low TCL might be appropriate.

Three major categories of considerations are used in determining a TCL: project, software, and test. Each column in figure 2-3 corresponds to one consideration. For example, the first column under "Project considerations" relates to cost; the second column under "Test considerations" relates to the comprehensiveness of testing required. For each column in the table, locate the description that best fits the software to be tested.

The first column describes four cost situations. If the most applicable description was found in the second box from the top of the column entitled "Normal cost constraints," then a TCL value of 1 would be appropriate for that consideration.

Evaluate a TCL for each column in the table. When completed, you should have eight TCL values. Compute the average of these eight values; the result is the final TCL. This value will be used in step 2. The process is illustrated in figure 2-4.

If one or more of the considerations have more weight than the others, then you may wish to compute a weighted average. For example, if the "Criticality" column is of primary importance, then you might weight it by some factor (e.g., a factor of 5). If no weights are needed, simply set all weighting factors to 1.

PROJECT CONSIDERATIONS			SOFTWARE CONSIDERATIONS			TEST CONSIDERATIONS			
COST	CRITICALITY	SCHEDULE	COMPLEXITY	DEVT FORMALITY	SOFTWARE CATEGORY	ERROR DETECTION	TEST COMPREHENSIVENESS	SOFTWARE EXAMPLES	CONFIDENCE LEVEL
LOW BUDGET EMPHASIS ON MIN COST	NO CRITICALITY ASSIGNMENT	TIGHT SCHEDULE	STRAIGHT-FORWARD SOLUTION, EASY TO CHECKOUT	FEW DEFINED REQUIREMENTS; INFORMAL DEVELOPMENT, USED LOCALLY	ONE-SHOT, PROTOTYPE, TEST S/W, DEMO S/W	PRESENCE OF RESIDUAL ERRORS NOT CONCERN	DETECT OBVIOUS ERRORS, S/W RESPONDS CORRECTLY TO NOMINAL CONDITIONS	TEST GENER; CONVERSION TABLE; TRADE STUDY SIMULATED	0
NORMAL COST CONSTRAINTS	NUISANCE IMPACT	SOME SCHEDULE CONSTRAINTS	MODERATE COMPLEXITY	NORMAL TO STRONG CONTRACTOR CONTROLS	GROUND BASED S/W; DATA REDUCTION, MISSION PREP S/W	RESIDUAL ERRORS NOTED FOR BLOCK CHANGE TEMP WORKAROUNDS USED	S/W REQUIREMENTS VALIDATED WIDE RANGE OF SAMPLE DATA	EDITOR, COMPILER, MISSION SIMULATION, ENVIRONMENTAL SIMULATOR	1
SOME COST FLEXIBILITY	MISSION IMPACT	NORMAL SCHEDULE CONSTRAINTS	GREATER COMPLEXITY	STRONG CONTRACTUAL CONTROLS FORMAL REVIEWS	REAL-TIME AVIONICS, C^3 S/W, C^3I	NO ERRORS THAT EFFECT MISSION ACCEPTABLE, RESIDUAL ERRORS REMOVED ASAP	S/W FUNCTIONS VALIDATED, DETECT ALL POSSIBLE ERRORS	AWACS, ALCM, PMALS, C^3I, AVIONICS MISSION PLANNING	2
COST NOT PREDOMINANT FACTOR; RELATIVELY UNCONSTRAINED	NUCLEAR; FLIGHT CREW SAFETY	ADDITIONAL ERROR DETECT REQUIREMENTS WILL NOT IMPACT SCHEDULE	DIFFICULT PROBLEM; COMPLEX SOLUTION; HARD TO TEST	GENERALLY CONTRACTED RIGID CONTROLS OVER DEVELOPMENT	HIGHLY CRITICAL APPLICATIONS; POSSIBLE CATASTROPHIC RESULTS	PRESENCE OF RESIDUAL ERRORS NOT ACCEPTABLE	DETECT ALL POSSIBLE ERRORS S/W DESIGN EXPLICITILY VALIDATED; COMPLETE STATE-OF-THE ART TESTING	NUCLEAR CONTROLS; CRITICAL SOFTWARE	3

Figure 2-3. Testing Confidence Level

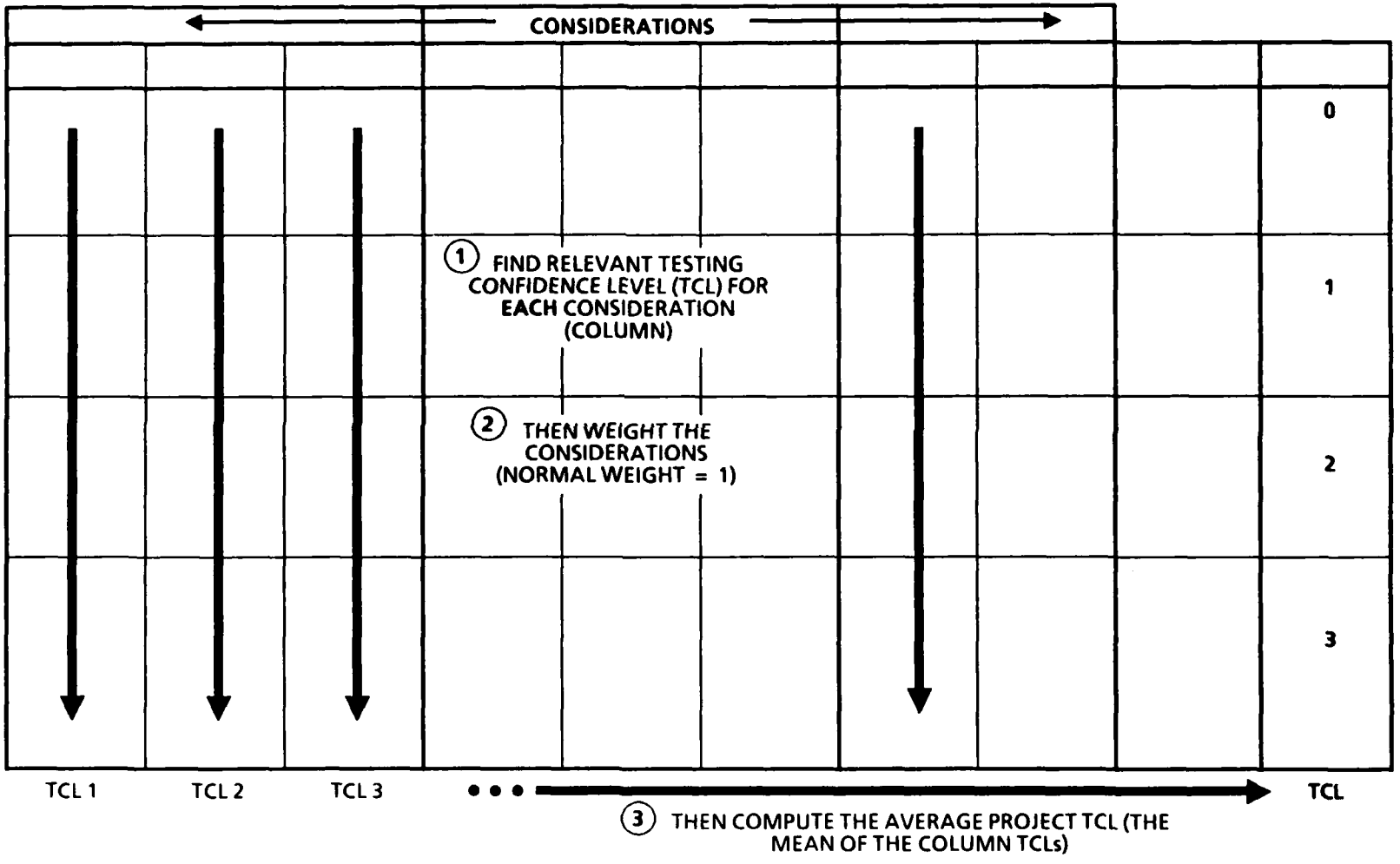


Figure 2-4. How To Compute Testing Confidence Level (TCL)

SOFTWARE TO BE TESTED PATH ONE EXAMPLE

CONSIDERATIONS	WEIGHT (1 = NORMAL)	TCL (0, 1, 2, 3) FIGURE 2-3	PRODUCT (WEIGHT X TCL)
COST	2	2	4
CRITICALITY	1	1	1
SCHEDULE	3	1	3
COMPLEXITY	1	2	2
DEVELOPMENT FORMALITY	1	1	1
SOFTWARE CATEGORY	1	1	1
ERROR DETECTION	1	1	1
TEST COMPREHENSIVENESS	1	1	1
SUM	11		14

$$TCL = \frac{\text{SUM OF PRODUCT}}{\text{SUM OF WEIGHT}} = \frac{14}{11} = 1.27$$

Figure 2-5. TCL Worksheet for Path 1

One of the considerations may be considered all important, and the remainder of the column TCL ratings would then be ignored. However, in such a case, determine all of the column TCL's to ensure that no important consideration is being overlooked. The computed TCL may not result in a whole number (0, 1, 2, 3), but instead in values such as 0.8, 1.2, 2.3, 1.7, etc. The recommended method of handling such results is to round the numbers toward the most highly weighted considerations. Figure 2-5 illustrates a completed TCL worksheet.

2.2.2 Step 2

Review the list of 18 software categories found in figure 2-7 and select the category most relevant. The categories contain groupings of similar complexity and criticality which were derived from a survey of Air Force testing practice and software engineering considerations. The results of these evaluations were used to recommend appropriate testing techniques and these appear in the tables used in the next step. If you find it difficult to identify an appropriate software category in figure 2-7, then refer to the appendix corresponding to your mission area and find the software function that most corresponds to the software being tested. Note the category number assigned to that software function. This category number is then used to identify the appropriate software category in figure 2-7.

2.2.3 Step 3

Using the TCL determined in step 1 and the software category chosen in step 2, use figure 2-9 to select candidate techniques as follows:

Enter the table at the left with the software category, reading to the right to select candidate techniques. It is important to select candidate techniques that are rated less than or equal to the TCL. For example, if the TCL = 1, then note all techniques that are rated 1 or 0. If the TCL = 2, then note all techniques that are rated 2, 1, or 0. If the TCL = 3, all techniques with ratings 3, 2, 1, and 0 are applicable in this extreme case. This is true because if the TCL is low, then only inexpensive, straightforward methods are appropriate and the table gives this result. If the TCL is very high, then you should use all of the basic techniques, augmented by the more expensive and esoteric techniques.

2.2.4 Step 4

Indicate the candidate techniques by entering an "X" in the first column on the Selection Worksheet as shown in figure 2-6. If possible, exercise path 2 and path 3 as described in sections 2.3 and 2.4 to add to or verify the list of candidate testing techniques.

A completed example problem can be found in section 2.8. Figure 2-18 shows a completed worksheet after using all three selection paths. Blank worksheets are provided in section 2.9.

The ratings in figure 2-9 were chosen to include as many candidate techniques as possible. This approach allows the user to select testing techniques from a wide range of candidates. The guidebook philosophy is that it is better to include an inappropriate or experimental technique in the candidate list – which the guidebook user may reject – than to exclude an esoteric technique that may be useful in rare instances.

SOFTWARE TO BE TESTED PATH ONE EXAMPLE

TCL 1.27 = 1.0 SOFTWARE CATEGORY (10) SENSOR & SIGNAL PROC.

	SOFTWARE TEST TECHNIQUES	PATH 1	PATH 2	PATH 3	NOTES/ COMMENTS
STATIC ANALYSIS	Code Reviews	✓			ROUNDED
	Error/Anomaly Detection	✓			TCL DOWN
	Structure Analysis/Documentation	✓			IN THIS
	Program Quality Analysis	✓			CASE...
	Input Space Partitioning				
	A. Path Analysis	✓			FROM 1.27
	B. Domain Testing	✓			TO 1.0
	C. Partition Analysis				
	Data-Flow Guided Testing				
DYNAMIC ANALYSIS	Instrumentation Based Testing				
	A. Path/Structural Analysis	✓			
	B. Performance Measurement	✓			
	C. Assertion Checking				
	D. Debug Aids	✓			
	Random Testing				
	Functional Testing	✓			
	Mutation Testing				
	Real-Time Testing	✓			
	SYMBOLIC TESTING				
	FORMAL ANALYSIS				

Figure 2-6. Selection Worksheet for Path 1

No.	Software category	Description
1.	Batch (general)	Can be run as a normal batch job and makes no unusual hardware or input-output actions (e.g., payroll program and wind tunnel data analysis program). Small, throwaway programs for preliminary analysis also fit in this category.
2.	Event control	Does real-time processing of data resulting from external events. An example might be a computer program that processes telemetry data.
3.	Process control	Receives data from an external source and issues commands to that source to control its actions based on the received data.
4.	Procedure control	Controls other software; for example, an operating system that controls execution of time-shared and batch computer programs.
5.	Navigation	Does computation and modeling to compute information required to guide an airplane from point of origin to destination.
6.	Flight dynamics	Uses the functions computed by navigation software and augmented by control theory to control the entire flight of an aircraft.
7.	Orbital dynamics	Resembles navigation and flight dynamics software, but has the additional complexity required by orbital navigation, such as a more complex reference system and the inclusion of gravitational effects of other heavenly bodies.
8.	Message processing	Handles input and output messages, processing the text or information contained therein.
9.	Diagnostic software	Used to detect and isolate hardware errors in the computer in which it resides or in other hardware that can communicate with that computer.

Figure 2-7 (Beginning)

No.	Software category	Description
10.	Sensor and signal processing	Similar to that of message processing, except that it requires greater processing, analyzing, and transforming the input into a usable data processing format.
11.	Simulation	Used to simulate an environment, mission situation, other hardware, and inputs from these to enable a more realistic evaluation of a computer program or a piece of hardware.
12.	Database management	Manages the storage and access of (typically large) groups of data. Such software can also often prepare reports in user-defined formats, based on the contents of the database.
13.	Data acquisition	Receives information in real-time and stores it in some form suitable for later processing; for example, software that receives data from a space probe and files it for later analysis.
14.	Data presentation	Formats and transforms data, as necessary, for convenient and understandable displays for humans. Typically, such displays would be for some screen presentation.
15.	Decision and planning aids	Uses artificial intelligence techniques to provide an expert system to evaluate data and provide additional information and consideration for decision and policymakers.
16.	Pattern and image processing	Used for computer image generation and processing. Such software may analyze terrain data and generate images based on stored data.
17.	Computer system software	Provides services to operational computer programs (i.e., problem oriented).
18.	Software development tools	Provides services to aid in the development of software (e.g., compilers, assemblers, static and dynamic analyzers).

Figure 2-7 (Concluded)

Notes:
The numbers in this table are related to the test confidence level (TCL) computed using table 2-3. Select testing techniques that are rated ≤ TCL.

Path/Structural Analysis rating based on branch testing

	STATIC ANALYSIS									DYNAMIC ANALYSIS										
TESTING TECHNIQUES / SOFTWARE CATEGORY	Code Reviews	Error/Anomaly Detection	Structure Analysis/Documentation	Program Quality Analysis	Input Space Partitioning	A Path Analysis	B Domain Testing	C Partition Analysis	Data-Flow Guided Testing	Instrumentation-Based Testing	A Path/Structural Analysis	B Performance Measurement	C Assertion Checking	D Debug Aids	Random Testing	Functional Testing	Mutation Testing	Real-Time Testing	SYMBOLIC TESTING	FORMAL ANALYSIS
BATCH (GENERAL)	0	1	2	2		2	1	2	3		2	1	2	2	3	0	3		3	3
EVENT CONTROL	0	0	1	1		1	1	3	3		2	1	2	1	2	0	3	2	3	3
PROCESS CONTROL	0	0	0	1		1	0	3	2		1	1	2	1	2	0	3	1	3	3
PROCEDURE CONTROL	0	0	0	0		1	0	3	2		1	1	1	0	2	0	2	1	2	3
NAVIGATION	0	0	0	2		2	2	3	2		2	2	2	2	3	0	3	1	3	3
FLIGHT DYNAMICS	0	0	0	1		2	1	2	2		2	2	2	2	2	0	3	0	3	3
ORBITAL DYNAMICS	0	0	0	1		2	1	1	2		2	2	1	0	2	0	2	1	2	3
MESSAGE PROCESSING	0	0	1	2		2	1	2	3		2	2	2	1	2	0	3	2	3	3
DIAGNOSTIC SOFTWARE	0	0	0	1		0	0	2	3		0	2	2	1	3	0	3	3	3	3
SENSOR & SIGNAL PROCESSING	0	0	0	0		0	0	2	2		1	1	2	0	2	0	2	1	3	3
SIMULATION	0	0	1	2		0	0	2	3		2	1	2	0	2	0	2	1	3	3
DATABASE MANAGEMENT	0	0	1	1		2	1	1	2		1	1	2	1	2	0	3		3	3
DATA ACQUISITION	0	0	1	2		2	0	3	3		2	1	2	1	2	0	3	1	3	3
DATA PRESENTATION	0	0	0	1		1	0	2	3		2	1	2	0	2	0	3	1	2	3
DECISION & PLANNING AIDS	0	0	0	1		1	1	1	2		1	2	2	1	2	0	3		2	3
PATTERN & IMAGE PROCESSING	0	0	1	1		1	0	1	2		1	2	2	0	2	0	3	1	2	3
COMPUTER SYSTEM SOFTWARE	0	0	0	1		1	1	1	2		1	2	2	0	2	0	2	1	2	3
SOFTWARE DEVELOPMENT TOOLS	0	0	0	1		1	1	1	1		1	2	2	0	2	0	3		3	

NOTE Blank entry indicates the testing technique is not applicable to the software category

Figure 2-8. Software Categories and Testing Techniques

2.3 PATH 2—TEST PHASE AND TEST OBJECTIVE

Figure 2-9 shows the steps of path 2, as described in this section.

2.3.1 Step 1

If path 1 has already been executed, then use the previously determined testing confidence level. If not, then determine the TCL using the confidence level table and the directions in section 2.2 which describes path 1, step 1.

2.3.2 Step 2

Use figure 2-11 to determine the appropriate testing techniques. Test phases, test objectives, and testing techniques were related in the table using state-of-the-art testing theory in conjunction with a survey of Air Force mission testing requirements. The correlation between test phases and objectives is given in figure 2-10. For example, one premise was that testing techniques used in unit and module test phases remain valid and useful in later phases. If the test phase is known, start at the far left of the table and locate the relevant test phase; then proceed to the right and identify each associated test objective. If only certain test objectives are known, locate those objectives in the center of the table.

2.3.3 Step 3

Proceed to the right and note the appropriate testing techniques for each of the associated test objectives.

It is important to select candidate techniques that are rated less than or equal to the TCL. For example, if the TCL = 1, then note all techniques that are rated 1 or 0. If the TCL = 2, then note all techniques that are rated 2, 1, or 0. If the TCL = 3, all techniques with ratings 3, 2, 1, and 0 are applicable in this extreme case. This is true because if the TCL is low, then only inexpensive, straightforward methods are appropriate and the table gives this result. If the TCL is very high, then you should use all of the basic techniques, augmented by the more expensive and esoteric techniques.

2.3.4 Step 4

Identify the candidate techniques by entering an "X" on the Selection Worksheet in column 2 for path 2. If possible, exercise path 1 and path 3 as described in sections 2.2 and 2.4 to add to or verify the list of candidate testing techniques. Figure 2-11 illustrates the worksheet after completion of Path 2.

A completed example problem can be found in Section 2.8. Figure 2-18 shows a completed worksheet after using all three selection paths. Blank worksheets are provided in Section 2.9.

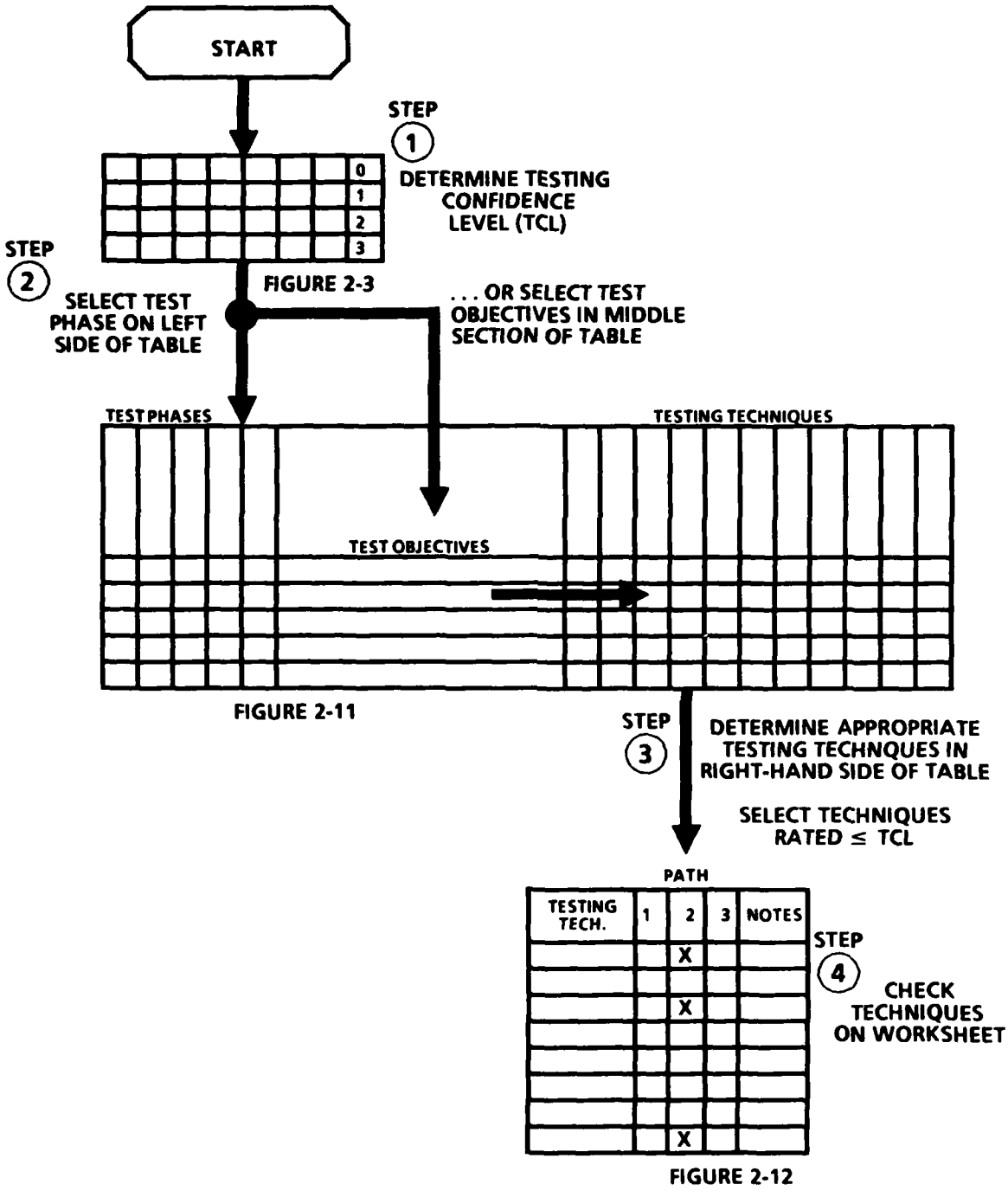


Figure 2-9. Path 2–Selection of Software Testing Techniques by Test Phase or Test Objective

This section contains the objectives of the test phases used in this guidebook as follows–

Test Phase	Objectives
Algorithm Confirmation	Verify that the proposed algorithm will satisfy all the stated and derived requirements of the software being designed. This phase may involve the selection of a "best" algorithm from a group of candidate approaches.
Design Verification	Verify that the design is a correct implementation of the specified requirements, and to review the quality and efficiency of the design.
Unit Test	Find discrepancies between the unit's logic and interfaces, and its design specifications (the descriptions of module's function, inputs, outputs, and external effects.) At this phase, compilation of the unit occurs. A unit is the smallest compilable entity of the computer program.
Module Test	Find discrepancies between the module's logic and interfaces, and the module design specifications (the descriptions of module's function, inputs, outputs, and external effects.) At this phase, compilation of the module occurs. In this context, a module is considered a combination of individual units that is smaller than the CPCI.
Integration Test	Combine and test several units and modules in order to check that the interfaces are defined correctly, and that the combined modules work correctly together. This process is done incrementally. Sometimes it proceeds top-down, with stubs used for the missing modules; sometimes integration is done bottom-up, starting with combinations of most basic modules. At present, the former is theoretically preferred, but the latter is still often performed.
Verification/CPCI Test	Verify that the computer program configuration item (CPCI) is a correct implementation of the specified design.

Figure 2-10 Test Phases and their Objectives (Beginning)

PQT/FQT	Perform a controlled execution of a deliverable program package such that all specified real-time and functional requirements are known to be satisfied. These tests are normally done according to government-approved test plans, such that completion of the FQT results in preliminary acceptance of the software by the government.
System Test	Find discrepancies between the system and its specifications, or to prove that the entire system meets it system level specifications. In this test, the software is integrated with the associated hardware.
Mission Test	Verify that the entire system correctly fulfills the mission for which it was intended. This test phase is designed to demonstrate that the system as specified meets the requirements of the mission.

Figure 2-10 Test Phases and their Objectives (Concluded)

Notes:
The numbers in this table are related to the test confidence level (TCL) computed using table 2-3. Select testing techniques that are rated ≤ TCL.

Path/Structural Analysis rating based on branch testing

TEST PHASES							STATIC ANALYSIS									DYNAMIC ANALYSIS											
ALGORITHM CONFIRMATION	DESIGN VERIFICATION/UNIT TEST MODULE TEST	INTEGRATION TEST	VERIFICATION/CPCI TEST	POT/FOT/SYSTEM TEST MISSION TEST	TEST OBJECTIVES	TESTING TECHNIQUES	Code Reviews	Error/Anomaly Detection	Structure Analysis/Documentation	Program Quality Analysis	Input Space Partitioning	A. Path Analysis	B. Domain Testing	C. Partition Analysis	Data-Flow Guided Testing	Instrumentation-Based Testing	A. Path/Structural Analysis	B. Performance Measurement	C. Assertion Checking	D. Debug Aids	Random Testing	Functional Testing	Mutation Testing	Real-Time Testing	SYMBOLIC TESTING	FORMAL ANALYSIS
	X	X	X	X	DETECT CODING ERRORS		0	0	0	2		1	1	1	3		0		1	0	1	1	2	2	3	
	X	X	X	X	DETECT PROGRAM LOGIC ERRORS		0	0	0	2		1	1	1	2		0	2		0	2	0	2	1	3	3
	X	X	X	X	DETECT OUTPUT FORMAT ERRORS		0	0	1	2		2	1	1			1			1						
	X	X	X	X	DETECT OUTPUT CONTENT ERRORS		0	1	1	2		2	0	0	3		1		2	1		0			3	3
	X	X	X	X	DETECT DESIGN ERRORS		0	0	0	2		1	0	0	2		0	2	2	1	2	0	2	1	3	3
	X	X	X	X	DETECT DESIGN DEFICIENCIES		0	0	0	2		1	1	1	2		0	2	3	0	1	0	3	1	3	3
	X	X	X	X	DETECT DATA ERRORS (CONTENT, FORMAT)		0	0	1			2	2	2	2		2		2	1	1	1	2	2	3	3
	X	X	X	X	DETECT DATA I/O ERRORS (TRANSFER, COMMUNICATION)		0	0	0			2	2	0			2		2	1		0	2	2		
	X	X	X	X	DETECT DATA DEFICIENCIES		0	0	2	2		2	2	2	3		2		3	1	2	1	2	2		
X	X	X	X	X	DETECT PROC. ACCURACY & PRECISION ERRORS/DEFICIENCIES		0	0	1			1	0	2	2		1	3	1	2	0		3			
	X	X	X	X	DETECT PERFORMANCE REQMTS ERRORS/DEFICIENCIES		0	0	1			1	1	1	2		1	1		1	1	0	3	2		
	X	X	X	X	DETECT FUNCTIONAL REQMTS ERRORS/DEFICIENCIES		0	1	1			1	0	0	3		1	1		1	1	0	3	1		
		X	X	X	VALIDATE FUNCTIONAL AND PERFORMANCE REQUIREMENTS		0	0	0			1	0	0	3		1	1		0	1	0	3	0		
X		X	X	X	VALIDATE DESIGN REQUIREMENTS		0	0	0			1	0	0	3		1	1		0	1	0	3	0		
X		X	X	X	VALIDATE SOFTWARE INTERFACE COMPATIBILITY		1	0	0			2	2	2	3		2			1	1	0	3	0		
			X	X	VALIDATE HARDWARE INTERFACE COMPATIBILITY		0	1	1								2			0			3	0		
X			X	X	VALIDATE ALGORITHMS/EQUATIONS		0	0	0			1	1	1			1	2		1		0				
				X	EVALUATE SOFTWARE PERFORMANCE CAPABILITIES							1	1	1			0	0		2	2	0		0		
				X	EVALUATE MISSION PERFORMANCE CAPABILITIES/DATA							1	1	1			0			2	2	0		0		
			X	X	VALIDATE SYSTEM OPERATION CAPABILITIES							1	1	0			0			2	2	0		0		
				X	EVALUATE SYSTEM INTERFACE COMPATIBILITY												0							1		
			X	X	VALIDATE PROGRAM DATA REQUIREMENTS							0	0	0			2		2			0		1	3	3

NOTE: Blank entry indicates the testing technique is not applicable to the test objective

Figure 2-11. Test Phase and Test Objectives/Testing Techniques

SOFTWARE TO BE TESTED PATH 2 EXAMPLE

TCL 1.0 SOFTWARE CATEGORY TEST OBJ: DETECT LOGIC ERROR

	SOFTWARE TEST TECHNIQUES	PATH 1	PATH 2	PATH 3	NOTES/ COMMENTS
STATIC ANALYSIS	Code Reviews		✓		
	Error/Anomaly Detection		✓		
	Structure Analysis/Documentation		✓		
	Program Quality Analysis				
	Input Space Partitioning				
	A. Path Analysis		✓		
	B. Domain Testing		✓		
	C. Partition Analysis		✓		
	Data-Flow Guided Testing				
DYNAMIC ANALYSIS	Instrumentation Based Testing				
	A. Path/Structural Analysis		✓		
	B. Performance Measurement		✓		
	C. Assertion Checking				
	D. Debug Aids		✓		
	Random Testing				
	Functional Testing		✓		
	Mutation Testing				
	Real-Time Testing		✓		
	SYMBOLIC TESTING				
	FORMAL ANALYSIS				

Figure 2-12. Selection Worksheet for Path 2

2.4 PATH 3--SOFTWARE ERROR CATEGORY

This path should be used if you know the categories of software errors that are occurring in the software to be tested, or that have occurred in similar projects and are likely to occur in this one. The categories of software errors used in this table are based on the categorization used in the Software Reliability Study (ref. THA76). Figure 2-13 shows the steps in path 3. Note that this path does not use the TCL at all.

2.4.1 Step 1

Locate the software error categories in table 2-14 that are either occurring or are predicted to occur. Note the software testing techniques that are rated as effective against these errors. Also note their relative effectiveness. The ratings are H (high), M (moderate), L (low) and are a measure of the effectiveness of a technique at detecting software errors in specific categories. Add these techniques to the candidate list. In most cases, choose only the most highly rated techniques in that row. If there are techniques rated H, use only those techniques; if the highest ratings are M, use those. However, if the highest rating is L, it is doubtful that the technique will have a significant effect against this error type. The effectiveness ratings are used to aid in the selection process. If an error category has an H in its row, then that technique is very effective at detecting that type of error. The H, M, L ratings are based only on this criterion. Other factors such as cost and ease of use are not included and must be considered separately (sec. 2.5). Program Quality Analysis may appear to be a very poor technique based on its very low rating against most error categories. This is because it is not an error detection technique, per se, but a software quality measure which serves a different, and valid, purpose.

2.4.2 Step 2

Record the candidate techniques by indicating their ratings on the Selection Worksheet in the third column for path 3 as shown in figure 2-15. If possible, exercise path 1 and path 2 as described in sections 2.2 and 2.3 to add to or verify the list of candidate testing techniques.

Refer to section 2.8 for a completed example problem that shows how to use path 3 in selecting software testing techniques. Blank worksheets are provided in section 2.9.

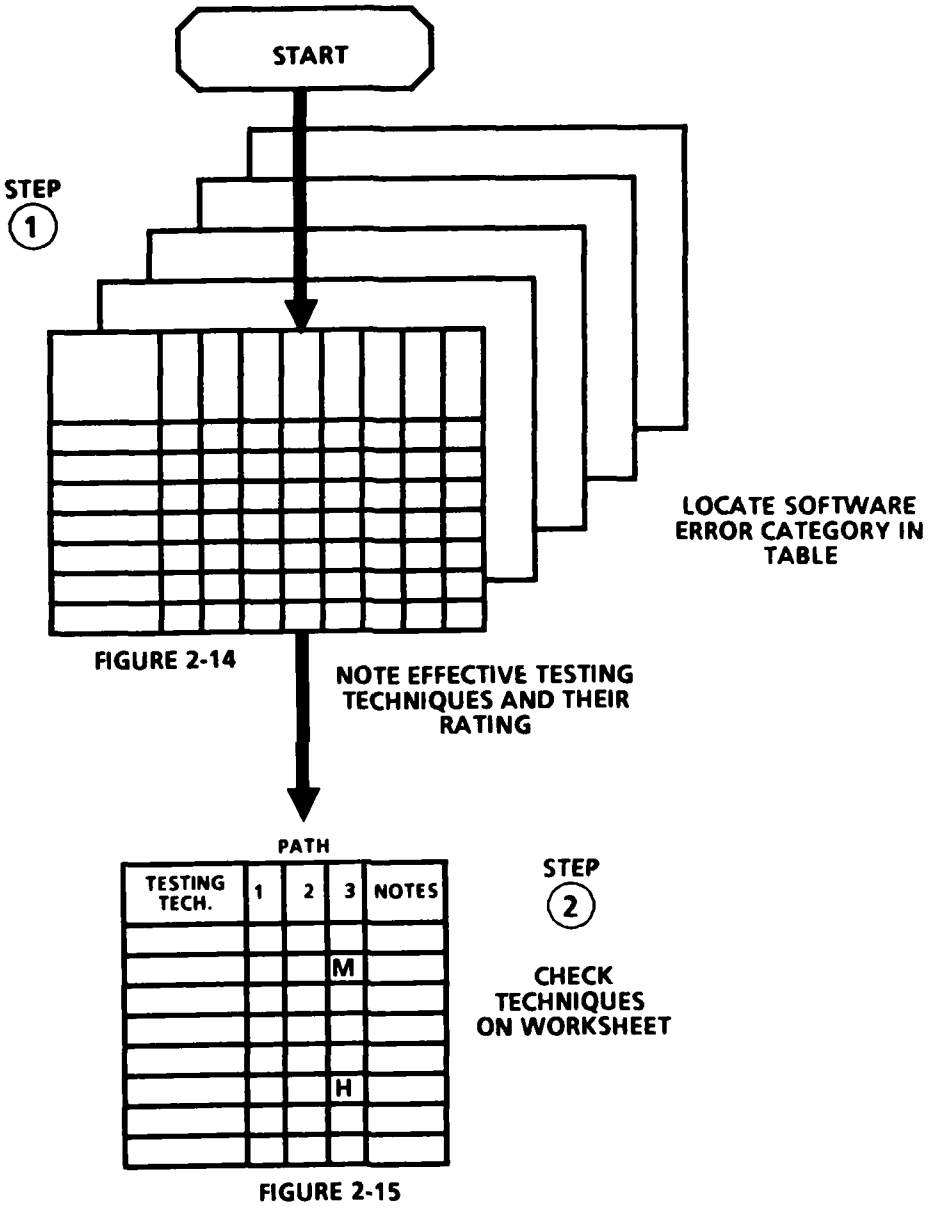


Figure 2-13. Path 3–Selection of Software Testing Techniques by Software Error Category

Note: Error detection ratings –
H - high M - medium L - low

		STATIC ANALYSIS									DYNAMIC ANALYSIS										
	TESTING TECHNIQUES	Code Reviews	Error/Anomaly Detection	Structure Analysis/Documentation	Program Quality Analysis	Input Space Partitioning	A. Path Analysis	B. Domain Testing	C. Partition Analysis	Data-Flow Guided Testing	Instrumentation-Based Testing	A. Path/Structural Analysis	B. Performance Measurement	C. Assertion Checking	D. Debug Aids	Random Testing	Functional Testing	Mutation Testing	Real-Time Testing	SYMBOLIC TESTING	FORMAL ANALYSIS
A-000	**COMPUTATIONAL ERRORS**																				
A-100	INCORRECT OPERAND IN EQUATION	H	H	L						M				M		L	M		L	L	M
A-200	INCORRECT USE OF PARENTHESIS	M	M	M										L	L	L	M		L	M	L
A-300	SIGN CONVENTION ERROR	H	M					L	L					H	L	L	M		L	M	
A-400	UNITS OR DATA CONVERSION ERRORS	H	H							L				L		L	M		L	L	
A-500	COMPUTATION PRODUCES AN OVER/UNDER FLOW													M		L	M		L	M	
A-600	INCORRECT/INACCURATE EQUATION USED	H													L	L	M	M		L	H
A-700	PRECISION LOSS DUE TO MIXED MODE	M	M							L						L	L		L		
A-800	MISSING COMPUTATION	M								H				M		L	H		L	M	M
A-900	ROUNDING OR TRUNCATION ERROR	L	L													L	L		L		
B-000	**LOGIC ERRORS**																				
B-100	INCORRECT OPERAND IN LOGICAL EXPRESSION	M	M					L		M						L	L	L	L	L	L
B-200	LOGIC ACTIVITIES OUT OF SEQUENCE	M	L				L	L	L	L		L		M	L	L	L		L	L	L
B-300	WRONG VARIABLE BEING CHECKED	M	M				M			H		M		L		L	L		L	L	L
B-400	MISSING LOGIC OR CONDITION TESTS	H	M				M		L			M		M	L	L	L		L	L	H
B-500	TOO MANY/FEW STATEMENTS IN LOOP	H	L	M	L					H		M		M		L	L		L	L	L
B-600	LOOP ITERATED INCORRECT # OF TIMES (INCLUDING ENDLESS LOOPS)	H					H	L	L	M		H	L	M	L	L	L		L	L	
B-700	DUPLICATE LOGIC	M	M	L	L		M			M		L		M		L	L		L	H	H

NOTE: Blank entry indicates the testing technique is not applicable to the error category

Table 2-14. Software Error and Testing Techniques

Note: Error detection ratings –
H - high M - medium L - low

		STATIC ANALYSIS									DYNAMIC ANALYSIS										
	TESTING TECHNIQUES	Code Reviews	Error/Anomaly Detection	Structure Analysis/Documentation	Program Quality Analysis	Input Space Partitioning	A. Path Analysis	B. Domain Testing	C. Partition Analysis	Data-Flow Guided Testing	Instrumentation-Based Testing	A. Path/Structural Analysis	B. Performance Measurement	C. Assertion Checking	D. Debug Aids	Random Testing	Functional Testing	Mutation Testing	Real-Time Testing	SYMBOLIC TESTING	FORMAL ANALYSIS
C-000	**DATA INPUT ERRORS**																				
C-100	INVALID INPUT READ FROM CORRECT DATA FILE	M	L							L				M		L	M		M		
C-200	INPUT READ FROM INCORRECT DATA FILE	M	L													L	M		M		
C-300	INCORRECT INPUT FORMAT	M	L													L	L		L		
C-400	INCORRECT FORMAT STATEMENT REFERENCED	M	L											L		L	L		L		
C-500	END OF FILE ENCOUNTERED PREMATURELY	L	L							L				L		L	M		M		
C-600	END OF FILE MISSING	L	L							L						L	M		M		
D-000	**DATA HANDLING ERRORS**																				
D-050	DATA FILE NOT REWOUND BEFORE READING	M	H							M		L			L	L	L		L	M	L
D-100	DATA INTIALIZATION NOT DONE	M	H							H		L		M	L	L	L		L	M	H
D-200	DATA INITIALIZATION DONE IMPROPERLY	M	L							M		L		L		L	L	L	L	M	L
D-300	VARIABLE USED AS A FLAG OR INDEX NOT SET PROPERLY	M	M							M		L		M		L	L	L	L	M	L
D-400	VARIABLE REFERRED TO BY THE WRONG NAME	M	H							M		L		M		L	L		L	L	H
D-500	BIT MANIPULATION DONE INCORRECTLY	M										L		L		L	L		L	L	
D-600	INCORRECT VARIABLE TYPE	M	H									L		M		L	L		L		
D-700	DATA PACKING/UNPACKING ERROR	M	M							L		L				L	L		L		
D-800	SORT ERROR	L	L							M		L				L	L		L	L	
D-900	SUBSCRIPTING ERROR	M	L							L		L				L	L	L	L		

NOTE: Blank entry indicates the testing technique is not applicable to the error category

Figure 2-14. (Continued)

Note: Error detection ratings –
H - high M - medium L - low

		STATIC ANALYSIS										DYNAMIC ANALYSIS											
TESTING TECHNIQUES		Code Reviews	Error/Anomaly Detection	Structure Analysis/Documentation	Program Quality Analysis	Input Space Partitioning	A. Path Analysis	B. Domain Testing	C. Partition Analysis	Data-Flow Guided Testing	Instrumentation-Based Testing	A. Path/Structural Analysis	B. Performance Measurement	C. Assertion Checking	D. Debug Aids	Random Testing	Functional Testing	Mutation Testing	Real-Time Testing	SYMBOLIC TESTING	FORMAL ANALYSIS
E-000	**DATA OUTPUT ERRORS**																				
E-100	DATA WRITTEN ON WRONG FILE	M	L												L	L	L		L		
E-200	DATA WRITTEN ACCORDING TO THE WRONG FORMAT STATEMENT	M	L												L	L		L			
E-300	DATA WRITTEN IN WRONG FORMAT	M	L												L	L	L		L		
E-400	DATA WRITTEN WITH WRONG CARRIAGE CONTROL	M	L												L	L	L		L		
E-500	INCOMPLETE OR MISSING OUTPUT	M	L													M	M		M	M	
E-600	OUTPUT FIELD SIZE TOO SMALL	L	L													M	M		M		
E-700	LINE COUNT OR PAGE EJECT PROBLEM	M	L									L				M	M		M		
E-800	OUTPUT GARBLED OR MISLEADING	L	L													M	M		M		
F-000	**INTERFACE ERRORS**																				
F-100	WRONG SUBROUTINE CALLED	M		M								H		L	L	L	L	L	L	L	
F-200	CALL TO SUBROUTINE NOT MADE OR MADE IN WRONG PLACE	M		M								H		M	L	L	L		L	L	
F-300	SUBROUTINE ARGUMENTS NOT CONSISTENT IN TYPE,UNITS,ETC	M	H	M						L		L		L		L	L	L	L		
F-400	SUBROUTINE CALLED IS NONEXISTENT	M	H	M								M				L	L		L	L	
F-500	SOFTWARE/DATA BASE INTERFACE ERROR	M	M	M						M		L			L	L	L		L		
F-600	SOFTWARE USER INTERFACE ERROR	M	L							L		M				L	L		L		
F-700	SOFTWARE/SOFTWARE INTERFACE ERROR	M	H	M								L		L		L	L		L		
G-000	**DATA DEFINITION ERRORS**																				
G-100	DATA NOT PROPERLY DEFINED/DIMENSIONED	M	H									M		L		L	L		L		
G-200	DATA REFERENCED OUT OF BOUNDS	M	M							M		L		M	L	L	L	L	L		
G-300	DATA BEING REFERENCED AT INCORRECT LOCATION	M	M							M		L		L	L	L	L		L		
G-400	DATA POINTERS NOT INCREMENTED PROPERLY	M	M							M		M			L	L	L	L	L	L	

NOTE: Blank entry indicates the testing technique is not applicable to the error category

Figure 2-14. (Continued)

Note: Error detection ratings –
H - high M - medium L - low

		STATIC ANALYSIS										DYNAMIC ANALYSIS											
	TESTING TECHNIQUES	Code Reviews	Error/Anomaly Detection	Structure Analysis/Documentation	Program Quality Analysis	Input Space Partitioning	A. Path Analysis	B. Domain Testing	C. Partition Analysis	Data-Flow Guided Testing	Instrumentation-Based Testing	A. Path/Structural Analysis	B. Performance Measurement	C. Assertion Checking	D. Debug Aids	Random Testing	Functional Testing	Mutation Testing	Real-Time Testing	SYMBOLIC TESTING	FORMAL ANALYSIS
H-000	**DATA BASE ERRORS**											L									
H-100	DATA NOT INITIALIZED IN DATA BASE	M	H									L		M		L	L		L		
H-200	DATA INITIALIZED TO INCORRECT VALUE	M								M		L		M		L	L	L	L		
H-300	DATA UNITS ARE INCORRECT	M	H													L	L		L		
I-000	**OPERATION ERRORS**																				
I-100	OPERATING SYSTEM ERROR (VENDOR SUPPLIED)											L					L		L		
I-200	HARDWARE ERROR																L		L		
I-300	OPERATOR ERROR											L					L		L		
I-400	TEST EXECUTION ERROR											L					L		L		
I-500	USER MISUNDERSTANDING/ERROR											L					L		L		
I-600	CONFIGURATION CONTROL ERROR																L		L		
J-000	**OTHER**																				
J-100	TIME LIMIT EXCEEDED												M			L	M		M		
J-200	CORE STORAGE LIMIT EXCEEDED												M			L	M		M		
J-300	OUPUT LINE LIMIT EXCEEDED												M			L	M		M		
J-400	COMPILATION ERROR	M	M	M																	
J-500	CODE OR DESIGN INEFFICIENT/NOT NECESSARY	H	M	M	M							M	M			L	M		M		
J-600	USER/PROGRAMMER REQUESTED ENHANCEMENT																				L
J-700	DESIGN NONRESPONSIVE TO REQUIREMENTS											M									L
J-800	CODE DELIVERY OR REDELIVERY																				
J-900	SOFTWARE NOT COMPATIBLE WITH PROJECT STANDARDS	M	M		H																

NOTE: Blank entry indicates the testing technique is not applicable to the error category

Figure 2-14. (Continued)

Note: Error detection ratings –
H - high M - medium L - low

	STATIC ANALYSIS										DYNAMIC ANALYSIS											
TESTING TECHNIQUES	Code Reviews	Error/Anomaly Detection	Structure Analysis/Documentation	Program Quality Analysis	Input Space Partitioning	A. Path Analysis	B. Domain Testing	C. Partition Analysis	Data-Flow Guided Testing	Instrumentation-Based Testing	A. Path/Structural Analysis	B. Performance Measurement	C. Assertion Checking	D. Debug Aids	Random Testing	Functional Testing	Mutation Testing	Real-Time Testing	SYMBOLIC TESTING	FORMAL ANALYSIS
L-000 REAL TIME ERRORS																				
L-100 ILLEGAL USE OF SHARED VARIABLES	M	M							H		L		L	L	L	M		M		
L-200 SYNCHRONIZATION ERRORS	M										L	L		L	L	M		H		
L-300 DEADLOCK	M								M		M	L			L	M		M		

NOTE: Blank entry indicates the testing technique is not applicable to the error category

Figure 2-14. (Concluded)

SOFTWARE TO BE TESTED PATH 3 EXAMPLE

TCL N/A SOFTWARE CATEGORY L-100(ERROR CATEGORY)

	SOFTWARE TEST TECHNIQUES	PATH 1	PATH 2	PATH 3	NOTES/ COMMENTS
STATIC ANALYSIS	Code Reviews			M	
	Error/Anomaly Detection			M	
	Structure Analysis/Documentation				
	Program Quality Analysis				
	Input Space Partitioning				
	A. Path Analysis				
	B. Domain Testing				
	C. Partition Analysis				
	Data-Flow Guided Testing			H	
DYNAMIC ANALYSIS	Instrumentation Based Testing				
	A. Path/Structural Analysis			L	
	B. Performance Measurement				
	C. Assertion Checking			L	
	D. Debug Aids			L	
	Random Testing			L	
	Functional Testing			M	
	Mutation Testing				
	Real-Time Testing			M	
	SYMBOLIC TESTING				
	FORMAL ANALYSIS				

Figure 2-15. Selection Worksheet for Path 3

2.5 GUIDELINES FOR FINAL SELECTION OF TESTING TECHNIQUES

If all three paths of testing technique selection are completed, the worksheet will be similar to the one shown in figure 2-18.

Before arriving at a final selection of techniques from the candidate list, the following points should be considered. These guidelines are general and each situation should be considered unique, in that no general set of guidelines can be effective in all cases. The judgments and evaluations are qualitative; it is not possible to provide firm guidelines and precise methods of evaluation. Discussions with experienced software test engineers who have used the selected techniques will prove valuable. Refer to section 3.0 to identify available automated testing tools and their sources.

2.5.1 Technique Suitability

Is the technique applicable to this specific environment? Are there any special considerations that make this testing especially suitable or completely invalid? What are the strengths of the technique in this environment and are they appropriate here?

2.5.2 Costs

The goal is to assemble the most effective testing techniques appropriate to the current software at the least cost.

2.5.3 Special Training

Decide whether the benefit of the technique justifies the extra training expense. Is there a less expensive alternative that is as effective in the same areas, but that does not require additional training?

2.5.4 Special Hardware

Will the additional hardware required by the technique result in enough improvement in test capability to justify its cost, or is there a less expensive alternative?

2.5.5 Special Software

Real-time software should be tested in a real-time simulator, if possible, before testing in the real-time environment. The simulator may make it possible to use techniques that could not be used in the real environment, because the simulator may provide additional main and mass storage or input/output facilities not available in the onboard or inflight environment. A simulator may also allow using dynamic techniques that could not be implemented in the real environment.

2.5.6 Input Requirements

Will the input requirements of the technique make it necessary to have special training or special hardware?

2.5.7 Output Requirements

Some techniques require a mass storage device for recording data. Other techniques require additional main memory storage. Assess the impact of the output requirements on the proposed test environment. Can the environment support the techniques being considered?

2.5.8 Candidate Testing Techniques

Do the candidate testing techniques complement each other, or do they have mutually conflicting operational requirements? Does one technique duplicate another without providing any new information in this environment?

2.6 SELECTION OF SUPPORT TOOLS

Identify and select automated support techniques using table 2.6-1 to complement the tools and techniques previously selected. The support techniques noted in this table are described in section 4.0. Also, appropriate support techniques are noted in some testing techniques descriptions.

		TEST DATA GENERATOR	TEST RESULT ANALYZER	TEST DOCUMENT WRITER	TEST MGMT SYSTEM	COMPLETION CRITERIA SOFTWARE	TEST DRIVER HARNESS
STATIC ANALYSIS	Code Reviews		X				
	Error/Anomaly Detection			X	X		
	Structure Analysis/Doc.			X	X		
	Program Quality Analysis			X	X		
	Input Space Partitioning						
	A. Path Analysis	X	X	X	X		
	B. Domain Testing	X	X	X	X		
	C. Partition Analysis	X	X	X	X		
	Data-Flow Guided Testing	X		X	X		
DYNAMIC ANALYSIS	Instrumentation-Based Testing						
	A. Path/Structural Analysis		X	X	X	X	X
	B. Performance Measurement		X	X	X		X
	C. Assertion Checking		X	X	X		X
	D. Debug Aids		X	X	X		
	Random Testing	X	X	X	X	X	X
	Functional Testing	X	X	X	X		X
	Mutation Testing	X	X	X	X	X	X
	Real-Time Testing	X	X	X	X	X	X
	SYMBOLIC TESTING			X	X		
	FORMAL ANALYSIS			X			

Figure 2-16. Testing Techniques and Support Techniques

The availability and sources of software tools that incorporate or support the selected support techniques can be determined by using the tool catalogs and guidelines of section 3.0.

2.7 TEST COMPLETION CRITERIA

Despite the concerted effort of many investigators in Government, in industry, and in academia, no absolute answer to the question "When has the software been tested enough?" has been found. Sections 2.7.1 through 2.7.5 discuss several possible approaches to determine test completion criteria. However, selection of an appropriate approach must rest with the reader. The problem is well recognized in the literature:

"One of the most difficult questions to answer when testing a program is determining when to stop, *since there is no way of knowing if the error just detected is the last remaining error.* In fact, in anything but a small program, it is unreasonable to expect that all errors will eventually be detected."

—Glenford Myers, "The Art of Software Testing."

"Testing can show the presence of errors, never their absence!"

—Edsger Dijkstra.

"Testing ends when the budget is exhausted; the rest is called maintenance."

—Unknown, quoted in many versions.

2.7.1 Approach 1—Test Until No Errors Remain

Myers has noted a major flaw with this approach: it encourages weak tests. That is, if the tester's job is done when his test program finds no more errors, he is not motivated to find errors and subconsciously will write test programs that will show the tested program to be error free. The tester will not be motivated to design "destructive" test cases that force the tested program to its design limits.

2.7.2 Approach 2—Test Until a Method Is Exhausted

An example of this approach might be the requirement to test until all logical paths in the

software have been executed and no errors remain. An alternative requirement might be to test the software until all boundary-value cases have proved to be error free. This technique is superior to the first approach, but it also has limits of effectiveness: it is not helpful in a test phase in which the methodology is not applicable, such as an operational test. Further, it is a subjective criterion, because there is no way to determine that the methodology is applied rigorously, nor that it is necessarily the most appropriate methodology.

2.7.3 Approach 3—Set Error Count Goals

This is a positive approach. If a module is to be tested until four errors are found, the testers will be highly motivated to design tests that find errors. This approach has several problems. The first is how to compute the number of errors to be detected. This involves getting an estimate of the total number of errors in the program and an estimate of the percentage of errors that feasibly can be detected by testing. The most common way of determining these numbers is by using historical data from the development of a similar software product. Another less precise approach is to use industrywide data.

2.7.4 Approach 4—Error Prediction Models

A more elaborate variation of approach 3 is to use one of several error-prediction models, or software reliability models. There have been a number of such models proposed in the technical literature. One of the best summaries is "A Guidebook for Software Reliability Assessment" (GOE81). A second, useful source is the publication, "Quantitative Software Models," (QUA79).

Some models require testing the software for a length of time and recording the elapsed time between detection of successive errors. Other models require recording computer execution time between detected errors. Still other models involve seeding known but secret errors in the computer program being tested, and then examining the ratio of detected seeded errors to the total number of known seeded errors. This approach presumes that the effectiveness of the tests against real and seeded errors is roughly the same.

No one model is effective in all environments. Indeed, some models are very sensitive to the assumptions in its derivation; failure of the real world to meet these assumptions will often result in erratic predictions. The most up-to-date and balanced appraisal of these techniques for predicting software errors is in (GOE81).

2.7.5 Approach 5—Plotting Errors

This approach plots the number of errors found each day, week, or month during the test phase. As the slope of the curve levels off, it is presumed that the test is approaching completion. Associated with this plot would be a second plot showing the number of uncorrected errors. If this number does not begin to decrease after the first plot has leveled off, the errors are not being corrected.

2.7.6 Summary

The best completion criterion is probably a combination of all of the approaches. For the early phases of testing, approach 2 would be most efficient, and approaches 3 and 4 would be appropriate additions in the later phases. Remember that no one model will work in all environments. The choice of models should be done with circumspection; (GOE81) provides details of the considerations involved.

2.8 EXAMPLE PROBLEM

This section presents an example of how to use this guidebook in selecting software testing techniques using the three paths and their tables found in sections 2.2 through 2.4. It also provides a step-by-step discussion of how the three paths were used.

In the example problem, the software to be tested is a small computer program used to control the fusing of a weapon. The Air Force mission is armament, the cost constraints are normal, the schedule is moderately tight, and the software is going to be developed by the Air Force using relatively informal controls. We want to confirm the algorithm used in the software at this phase of the test, and because of the seriousness of its function, a thorough test is required. Finally, external considerations force us to be very concerned about any possible cost overrun. Past experience with projects of this kind have uncovered many software problems resulting from logical activities out of sequence.

Using the information in this example problem, we will follow all three paths of the guidebook to determine the appropriate software testing techniques.

2.8.1 Path 1

The first step is to evaluate the testing confidence level (TCL) appropriate for this situation, using figure 2-3. Figure 2-17 shows the TCL worksheet resulting from this step. Each column in the table concerns one consideration, and each consideration is rated and given a relative weight. Note that although the cost constraint was normal (= 1), this consideration has been weighted by a factor of 3 to reflect the external consideration of cost overrun. The development formality consideration was weighted 0 in this example, an unusual action taken only to show that it is a valid option. All other considerations were weighted normally, with a weight of 1.

The sum of the weights is 9, the weighted sum is 11. The weighted average is 1.22. We must next consider whether to round the quotient up or to round it down. Since the consideration with the most weight (cost) was originally scored as a 1, it is appropriate in this case to round down. We chose to round toward the rating of the heavily weighted consideration. Thus, we have TCL = 1. However, this is not a firm rule; common sense and sound judgement are the final arbiters in selecting the TCL. The second step in path 1 is to choose the appropriate software category. We first look at figure 2-7 and are not able to decide where our fusing software fits, or we want to confirm our category selection by another method. Since the software is in the armament mission, we look in appendix A, find the software type "fusing," and note that its category number = 2. This corresponds to the "event control" category in figure 2-7.

Now we are ready to use figure 2-8 to select software testing techniques based on the software category and the TCL. Since the category was "event control," we start at the second row of the table and proceed across, noting all techniques with a rating of 1 or 0. That is, we choose all techniques with a rating less than or equal to the computed TCL. The first six techniques are applicable because they are all rated 0 or 1, so we note these in the "Path 1" column of the worksheet. The next three techniques are rated 2 or 3 and are excluded. The entire row is examined, and the results noted in the remainder of the Path 1 column of the worksheet. We have now completed path 1.

SOFTWARE TO BE TESTED FUSING CONTROL

CONSIDERATIONS	WEIGHT (1 = NORMAL)	TCL (0, 1, 2, 3) FIGURE 2-3	PRODUCT (WEIGHT X TCL)
COST	3	1	3
CRITICALITY	1	2	2
SCHEDULE	1	1	1
COMPLEXITY	1	0	0
DEVELOPMENT FORMALITY	0	1	0
SOFTWARE CATEGORY	1	2	2
ERROR DETECTION	1	2	2
TEST COMPREHENSIVENESS	1	1	1
SUM	9		11

$$TCL = \frac{\text{SUM OF PRODUCT}}{\text{SUM OF WEIGHT}} = 1.22$$

Figure 2-17. Example TCL Worksheet

2.8.2 Path 2

Since we have already computed a TCL, we can go directly to figure 2-11. Our primary concern for this testing phase was to confirm the algorithms used in the fusing software. We first look in the left-hand portion of the table and find the phase "algorithrn confirmation" listed. Scanning down this column, we find four X's that mark the rows we will examine in the right-hand side of the table. Once again, we note only those testing techniques rated less than or equal to the chosen TCL; that is, 1 or 0. All these techniques are noted in the second column of the worksheet. We have now cornpleted path 2.

2.8.3 Path 3

Our example noted an error pattern in previous software of this type; namely, logical activities out of sequence. When we scan down figure 2-14, error category B-200 is found to closely describe this type of error. Scanning across the row, we find that no technique is rated highly (H) against this type of error. We decide to consider all techniques rated medium (M) and note these in the "Path 3" column of the worksheet.

2.8.4 Final Technique Selection

Now the entire worksheet is complete, as shown in figure 2-18. It contains a list of candidate testing techniques. Based on this worksheet, the next step is to review the technique descriptions in section 4.0 and to work through the consideration guidelines of section 2.5 to make a final selection. There are no further tables or guidelines to aid the user of the guidebook in the final selection of testing techniques for his unique environment. The user should remember that the guidebook is an aid in selecting software testing techniques, not an absolute authority. With the guidebook, the user may be prompted to consider points that might have been overlooked.

2.8.5 Identification of Test Tools

The user must now look up the techniques included in his final choice in one of the tool catalogs referenced in section 3.0 to determine the availability of automated tools that implement the chosen techniques. Further judgments are required of the user in section 3.0, and the guidelines of section 2.5 should be considered in this process.

SOFTWARE TO BE TESTED FUSING CONTROL

TCL 1.22 SOFTWARE CATEGORY 2

	SOFTWARE TEST TECHNIQUES	PATH 1	PATH 2	PATH 3	NOTES/ COMMENTS
STATIC ANALYSIS	Code Reviews	X	X	M	
	Error/Anomaly Detection	X			
	Structure Analysis/Documentation	X			
	Program Quality Analysis	X			
	Input Space Partitioning				
	A. Path Analysis	X	X		
	B. Domain Testing	X	X		
	C. Partition Analysis		X		
	Data-Flow Guided Testing				
DYNAMIC ANALYSIS	Instrumentation Based Testing				
	A. Path/Structural Analysis		X		
	B. Performance Measurement	X	X		
	C. Assertion Checking		X	M	
	D. Debug Aids		X		
	Random Testing	X	X		
	Functional Testing		X		
	Mutation Testing	X			
	Real-Time Testing		X		
	SYMBOLIC TESTING				
	FORMAL ANALYSIS				

Figure 2-18. Example Selection Worksheet

The guidebook example in this section is simplified to illustrate the use of the guidebook. Real problems will almost always be more nebulous, but the steps in choosing software testing techniques will remain essentially the same.

2.9 Blank Worksheets

Figure 2-19 is a blank TCL worksheet, figure 2-20 is a blank Selection Worksheet. These worksheets are provided as masters for reproducing working copies. Do not remove them from the guidebook so that they may continue to be available as reproduction masters.

SOFTWARE TO BE TESTED _____

CONSIDERATIONS	WEIGHT (1 = NORMAL)	TCL (0, 1, 2, 3) FIGURE 2-3	PRODUCT (WEIGHT X TCL)
COST			
CRITICALITY			
SCHEDULE			
COMPLEXITY			
DEVELOPMENT FORMALITY			
SOFTWARE CATEGORY			
ERROR DETECTION			
TEST COMPREHENSIVENESS			
SUM			

$$TCL = \frac{\text{SUM OF PRODUCT}}{\text{SUM OF WEIGHT}} =$$

Figure 2-19. TCL Worksheet

SOFTWARE TO BE TESTED _____

TCL _____ SOFTWARE CATEGORY _____

	SOFTWARE TEST TECHNIQUES	PATH 1	PATH 2	PATH 3	NOTES/ COMMENTS
STATIC ANALYSIS	Code Reviews				
	Error/Anomaly Detection				
	Structure Analysis/Documentation				
	Program Quality Analysis				
	Input Space Partitioning				
	A. Path Analysis				
	B. Domain Testing				
	C. Partition Analysis				
	Data-Flow Guided Testing				
DYNAMIC ANALYSIS	Instrumentation Based Testing				
	A. Path/Structural Analysis				
	B. Performance Measurement				
	C. Assertion Checking				
	D. Debug Aids				
	Random Testing				
	Functional Testing				
	Mutation Testing				
	Real-Time Testing				
	SYMBOLIC TESTING				
	FORMAL ANALYSIS				

Figure 2-20. Selection Worksheet

3.0 AVAILABILITY OF SOFTWARE TESTING TOOLS

3.1 CROSS-REFERENCES

This guidebook cross-references three catalogs of available software tools. Each catalog differs in the facilities offered and in the categories used to classify software tools. Two of the catalogs are available commercially; the third is maintained by the Government. These catalogs are listed as follows:

- "RCI Software Tools Directory."
- "SRA Software Engineering Automated Tools Index".
- "Software Development Tools." (NBS)

Tables 3.2-1, 3.3-1, and 3.4-1 cross-reference the categories within the guidebook testing techniques taxonomy to the tool categories used in the tool catalogs. When there is no identical category in the tool catalog corresponding to the guidebook category, the table will indicate the catalog categories in which the tool is most likely found. In some cases, no corresponding category is found and the table entry for the catalog is left blank.

Software tools are available from the commercial suppliers listed in the catalogs, and from the Government. Within the Government there are two principal sources:

a. Federal Software Exchange Center (FSEC)
NTIS Computer Products
5285 Port Royal Road
Springfield, Virginia 22161
(703) 487-4848 or FTS 737-4848

b. Language Control Facility (ASD/ADOL)
Wright-Patterson AFB, Ohio 45433
(513) 225-4472

3.2 RCI SOFTWARE TOOLS DIRECTORY

Published by:
Reifer Consultants, Inc. (RCI)
25550 Hawthorne Blvd.
Torrance, California 90505
(213) 373-8728
Cost (Oct. 83): $225.00

The February 1983 edition of the RCI catalog was reviewed. This catalog is available in a multivolume looseleaf binder set and is updated semiannually. In June 1983, RCI announced an online computerized version of the directory with tool category, keyword, and tool feature search capabilities. RCI also offers a tool evaluation service to its users; in June 1983, an automated version of this service was announced.

The RCI directory allows the reader to locate tools by the following major categories:

- Hardware.
- Vendor.
- Life cycle phase.
- Descriptive keywords.
- Major function.
- Specific capabilities.
- Input to tool.
- Output from tool.
- Function of tool.
- Supplier information.
- Source of information (for directory listing).

For each tool in the directory, there is a brief descriptive paragraph and information on—

- Machine(s).
- Operating environments.
- Number of users.
- Language.

Each tool is categorized as—

- Batch.
- Interactive.
- Real-time.

The status of each tool is categorized as—

- Experimental.
- Production.
- Production (no support).

There is a keyword glossary in the RCI directory with 79 keywords that is helpful in locating appropriate tools.

The following list contains the table of contents from the RCI directory.

7 TOOL SYSTEMS

7.1 General Purpose Systems

7.2 Application Development Systems

Table 3.2-1 correlates the tool categories in this guidebook to the relevant sections in the RCI directory. Since these categories are very broad and test tools are not divided into subcategories, the table also gives the relevant keywords (by number).

	TESTING TECHNIQUES	EQUIVALENT RCI TOOL FUNCTION CATEGORY	KEYWORD
STATIC ANALYSIS	Code Reviews		
	Error/Anomaly Detection	4.3, 4.4	6, 14, 15, 24, 40, 41, 54, 64
	Structure Analysis/Documentation	4.5	14, 24, 27, 33, 40, 40, 41, 52, 54, 65, 66, 69
	Program Quality Analysis	4.5, 5.2	54
	Input Space Partitioning	4.5	3, 54
	A. Path Analysis	4.5	3, 54
	B. Domain Testing	4.5	3, 35, 54
	C. Partition Analysis	4.5	3, 35, 54
	Data-Flow Guided Testing	4.5	20, 21
DYNAMIC ANALYSIS	Instrumentation-Based Testing	4.4, 4.5	
	A. Path/Structural Analysis	4.5	4, 54
	B. Performance Measurement	4.5, 5.2	4, 40, 50, 53, 54, 62
	C. Assertion Checking	4.2	4
	D. Debug Aids	4.5, 5.2	4, 14, 24, 40, 54
	Random Testing	4.5	35, 54
	Functional Testing	4.5	4, 26, 31, 34, 39, 54
	Mutation Testing	4.5	
	Real-Time Testing	4.5	26, 28, 29, 31, 34, 39, 54
	SYMBOLIC TESTING	4.5	70
	FORMAL ANALYSIS	4.5	70

Table 3.2-1. RCI Tool Directory Cross-Reference

3.3 SRA SOFTWARE ENGINEERING AUTOMATED TOOLS INDEX

Published by:
Software Research Associates (SRA)
P.O. Box 2432
San Francisco, California 94126
(415) 957-1441
Cost (Oct. 83): $225.00

This catalog is available in looseleaf binder form and is available as a computer-accessible database. An update service is available to keep the index current. The following description and the cross-reference table 3.3-1 are based on the January 1983 updates to the index.

The SRA index categorizes the tools by—

- Product name.
- Tool category (see following paragraph).
- Supplier.

The tools in the SRA index are classified according to a scheme which associates a special category number for each major class of tool. Generally, the categories reflect the position in which a software tool would be used in the software life cycle. The major categories are listed as follows, showing the subcategories of testing tools. (The other major categories also have between 2 and 12 subcategories.)

0.0 Software Tool Indexes
1.0 Requirement/Specification Tools
2.0 Software Design Tools
3.0 Software Implementation Tools
4.0 Software Testing Tools
4.1 Static Analyzer
4.2 Execution Verifier
4.3 Test File Generator
4.4 Test Data Generator
4.5 File Comparator
4.6 Test Pattern Generator

4.7 Test Bed System (Test Harness)
4.8 Compiler Validation System
4.9 Electronic data processing (EDP) Auditing Package
4.10 Program Proving System
4.11 Mutation System
4.12 Symbolic Evaluation System
5.0 Software Maintenance Tools
6.0 Software Project Management Tools
7.0 Languages and Language Processing Systems
8.0 Utility Packages
9.0 Miscellaneous
10.0 Research and Development Systems (Future Systems)

Table 3.3-1 correlates the guidebook taxonomy of software testing techniques to the categories in the SRA index.

3.4 SOFTWARE DEVELOPMENT TOOLS (NBS)

Published by:
DACS, RADC/ISISI
Griffiss AFB, N.Y. 13441
(315) 336-0937, Autovon 587-3395
Cost: not set as of Oct. 83

This catalog was originally prepared by Raymond C. Houghton, Jr., National Bureau of Standards, Special Publication 500-88, issued March 1982. The Data and Analysis Center for Software (DACS), a DOD Information Analysis Center, will periodically update the tool catalog. DACS is currently operated by the IIT Research Institute for RADC. DACS also offers a custom software tool search, which is an automated search of the DACS/NBS Tools Database, combined with manual searches of other tool directories.

This catalog provides many indexes to its tool list. The tools are indexed by–

- General classification:
  - Software management, control, and maintenance tools.
  - Software modeling and simulation tools.

	TESTING TECHNIQUES	EQUIVALENT SRA TOOL FUNCTION CATEGORIES
STATIC ANALYSIS	Code Reviews	2.0 SOFTWARE DESIGN TOOLS
	Error/Anomaly Detection	3.6 STD ENFORCER 3.1 DEBUGGING 4.1 STATIC ANALYZER 5.3 DATA DICTIONARY
	Structure Analysis/Documentation	5.7 INTERFACE 5.1 FLOWCHART GENERATOR 5.2 CROSS-REFERENCE
	Program Quality Analysis	4.1 STATIC ANALYZER
	Input Space Partitioning	4.0 SOFTWARE TESTING TOOLS
	A. Path Analysis	4.2 EXECUTION VERIFIER
	B. Domain Testing	4.3 TEST FILE 4.4 TEST DATA
	C. Partition Analysis	4.3 TEST FILE 4.4 TEST DATA
	Data-Flow Guided Testing	4.0 SOFTWARE TESTING TOOLS
DYNAMIC ANALYSIS	Instrumentation-Based Testing	3.0 SOFTWARE IMPLEMENTATION 4.0 SOFTWARE TESTING TOOLS
	A. Path/Structural Analysis	3.4 STRUCTURED 3.8 TRACE
	B. Performance Measurement	3.8 TRACE 4.2 EXECUTION VERIFIER 3.2 PERFORMANCE TUNER
	C. Assertion Checking	4.0 SOFTWARE TESTING TOOLS 3.0 SOFTWARE IMPLEMENTATION
	D. Debug Aids	3.1 DEBUG
	Random Testing	4.0 SOFTWARE TESTING TOOLS
	Functional Testing	4.7 TEST BED
	Mutation Testing	4.11 MUTATION SYSTEM
	Real-Time Testing	4.7 TEST BED
	SYMBOLIC TESTING	4.12 SYMBOLIC EVALUATION SYSTEM
	FORMAL ANALYSIS	4.10 PROGRAM PROVING

Table 3.3-1. SRA Tool Index Cross-Reference

    - Requirements and design specification and analysis tools.
    - Source program analysis and testing tools.
    - Software support system/programming environment tools.
    - Program construction and generation tools.

- Input subject (text, data, specific languages).
- Transformation features.
- Static analysis features.
- Dynamic analysis features.
- User output (diagnostics, graphics, listings).
- Machine output (specific languages).
- Portability.
- Source language.
- Hardware (specific computer models).
- Software (specific operating systems).
- Public domain.
- Information source.

The tool abstracts (or tool lists) contain the following categories (not all information is available for all tools).

- Acronym of tools.
- Title of tool.
- Classification (see above).
- Features (uses tool feature taxonomy developed for this catalog).
- Stage of development (concept, design, implemented).
- Date (of development).
- Implementation language.
- Tool portable (yes, no).
- Tool size.
- Computer (and other hardware).
- OS (operating system and other software required).
- Tool available.
- Public domain.
- Restrictions (copyrights, licenses).
- Tool support (if so, and by whom).

- Tool summary (brief, one-paragraph description).
- Documentation available and length.
- References.
- Developer(s).
- Contact.
- Information source (for this listing).

Table 3.4-1 lists the tool function categories in the NBS catalog that correspond to the testing technique categories used in this guidebook.

	TESTING TECHNIQUES	EQUIVALENT NBS FUNCTION CATEGORIES
STATIC ANALYSIS	Code Reviews	————
	Error/Anomaly Detection	STATIC ANALYSIS: CONSISTENCY, DATA FLOW, AUDITING, INTERFACE ANALYSIS
	Structure Analysis/Documentation	STATIC ANALYSIS: STRUCTURE CHECKING, SCANNING, COMPLETENESS, UNITS CHECKING
	Program Quality Analysis	STATIC ANALYSIS: COMPLEXITY
	Input Space Partitioning	STATIC ANALYSIS: ————
	A. Path Analysis	STATIC ANALYSIS: COVERAGE ANALYSIS
	B. Domain Testing	STATIC ANALYSIS: CONSTRAINT EVALUATION
	C. Partition Analysis	STATIC ANALYSIS: CONSTRAINT EVALUATION
	Data-Flow Guided Testing	STATIC ANALYSIS: DATA FLOW ANALYSIS
DYNAMIC ANALYSIS	Instrumentation-Based Testing	DYNAMIC ANALYSIS
	A. Path/Structural Analysis	DYNAMIC ANALYSIS: COVERAGE ANALYSIS, TRACING (LOGIC, DATA)
	B. Performance Measurement	DYNAMIC ANALYSIS: TUNING, RESOURCE UTILIZATION
	C. Assertion Checking	DYNAMIC ANALYSIS: ASSERTION CHECKING
	D. Debug Aids	DYNAMIC ANALYSIS: TRACING
	Random Testing	DYNAMIC ANALYSIS: ————
	Functional Testing	DYNAMIC ANALYSIS: CONSTRAINT EVALUATION, COVERAGE ANALYSIS
	Mutation Testing	DYNAMIC ANALYSIS: ————
	Real-Time Testing	DYNAMIC ANALYSIS: SIMULATION
	SYMBOLIC TESTING	DYNAMIC ANALYSIS: SYMBOLIC EXECUTION
	FORMAL ANALYSIS	————

Table 3.4-1. NBS Catalog Cross-Reference

3.5 ADDITIONAL SOURCES OF INFORMATION

3.5.1 OTHER CATALOGS

Several other catalogs of available software tools follow.

"COSMIC"
112 Barrow Hall
University of Georgia
Athens, Georgia 30602
(404) 542-3265

COSMIC (Computer Software Management and Software Organization) is a catalog of selected computer programs, published on microfiche and magnetic tape yearly by NASA. COSMIC contains over 1,300 computer programs in its library.

"Federal Software Exchange Catalog"
Federal Software Exchange
Center (FSEC)
NTIS Computer Products
5285 Port Royal Road
Springfield, Virginia 22161
(703) 487-4848 or
FTS 737-4848

Published once a year with two semiannual supplements published between catalog editions. Relevant sections of the 1983 edition are "Software Tools" and "Computer Sciences." The FSEC is a distributor of software development tools and associated documentation.

"ICP Software Directory"
International Computer Programs, Inc.
9000 Keystone Crossing
Indianapolis, Indiana 46240

International Computer Programs, Inc. (ICP) publishes this directory semiannually in January and July. In the January 1982 issue, the relevant sections are section 12, particularly 12.2, "Testing and Emulation," and section 14, particularly 14.3.3, "Application Testing." Each tool is described in a paragraph, and for each tool the following information is provided: product name, product type, geographical area served, hardware supported, languages, number of users, contact (source).

"Datapro Directory of Software"
Datapro Research Corporation
1805 Underwood Blvd.
Delran, New Jersey 08075

Each tool is described with a brief paragraph plus entries for the source company, functions, hardware systems, operating systems, pricing, and maintenance. The catalog is updated incrementally. The "Test and Debugging" section (D80-500-001) reviewed for this guidebook is dated February 1983.

"Software"
Data Decisions
20 Brace Road
Cherry Hill, New Jersey 08034

This catalog is updated in sections periodically; the January 1983 issue of Section 660, "Programming Support," contained listings of tools relevant to this guidebook.

3.5.2 Other References

In addition, the following references are especially recommended. They can be augmented, as necessary, by the bibliography.

"Today's Software Tools Point to Tomorrow's Tool Systems"
Electronic Design Magazine

This article was published in "Electronic Design," dated July 23, 1981, Vol 30, No. 15.

"A Review of Software Maintenance Technology"
U.S. Department of Commerce
National Technical
Information Service
5285 Port Royal Road
Springfield, Virginia 22151

This publication is RADC TR-80-13, NTIS Accession No. A083-985, dated February 1980.

"DoD Tools Directory"
Litton Mellonics Information Center
Litton Guidance & Control Systems
Mail Stop 50
5500 Canago Avenue
Woodland Hills, CA 91365

This is an internal computer listing containing primarily JOVIAL J73 tools maintained by Litton. Contact Mr. Howard E. Verne (213) 715-2931 for further information.

4.0 STATE-OF-THE-ART SOFTWARE TEST TECHNIQUES

4.1 INTRODUCTION

This section provides an introduction to state-of-the-art software test techniques. Section 4.2 contains a summary description of the taxonomy of test techniques. A more detailed description of the taxonomy is provided in section 4.3. It is advised to read section 4.2 prior to reading section 4.3. Section 4.4 contains descriptions of support techniques, which are used as aids to the test techniques. Section 4.5 contains brief descriptions of testing methods that are considered important, but are beyond the scope of this taxonomy since they treat activities that occur at the front-end of the life cycle. The final section contains the bibliography.

4.2 SUMMARY DESCRIPTIONS OF THE TAXONOMY

Summary descriptions of the taxonomy of testing techniques used in this handbook are provided in this section. Table 4.2-1 contains the categories of the taxonomy. Use this table as a guide to locating technique descriptions.

STATIC ANALYSIS
- Code Reviews and Walkthroughs
- Error and Anomaly Detection
- Structure Analysis and Documentation
- Program Quality Analysis
- Input Space Partitioning
  - Path Analysis
  - Domain Testing
  - Partition Analysis

Data-Flow Guided Testing

DYNAMIC ANALYSIS
- Instrumentation-Based Testing
  - Path and Structural Analysis
  - Performance Measurement
  - Assertion Checking
  - Debug Aids
- Random Testing
- Functional Testing
- Mutation Testing
- Real-Time Testing

SYMBOLIC TESTING

FORMAL ANALYSIS

Table 4.2-1. Taxonomy of Testing Techniques

The four main categories of the taxonomy are static analysis, dynamic analysis, symbolic testing, and formal analysis. Static analysis involves detecting errors by examining the software rather than by executing it. In contrast, dynamic analysis methods detect errors by executing the software. Symbolic testing involves executing the program symbolically by deriving mathematical expressions for the outputs of a program in terms of the inputs.

Overall, formal analysis methods are the least developed and therefore, the least used category of the taxonomy. The purpose of formal analysis is to apply the formality and rigor of mathematics to the task of proving the consistency between an algorithm solution and a rigorous, complete specification of the intent of the solution. The following four sections contain summary descriptions of the four main categories of the taxonomy.

4.2.1 Static Analysis

Static analysis detects errors by examining the software system (i.e., requirements statement, program code, users manual) rather than by executing it. Some examples of the errors detected are language syntax errors, misspellings, incorrect punctuation, improper sequencing of statements, and missing specification elements. Static analysis techniques may be manually or automatically applied, although automated techniques require a machine-readable specification of the product. Figure 4.2-1 provides a picture of the general form of static analysis.

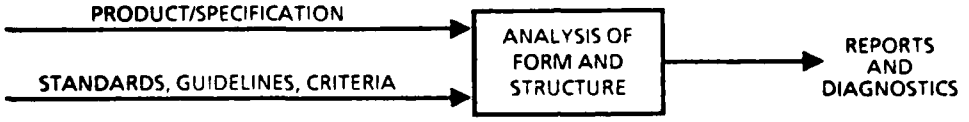


Figure 4.2-1. General Form of Static Analysis

Manual static analysis techniques may be applied to all development products such as the requirements statement, program code, or a users manual. In general, these techniques are straightforward and when applied with discipline are effective in preventing and detecting errors.

Application of certain manual techniques, such as desk checking, inspection, and walk-throughs, provide certain advantages over using specialized automated techniques. One advantage is that different perspectives can be addressed simultaneously. A product may be examined for high-level and detailed properties. Another advantage is that manual analysis provides an opportunity for the analyst to apply various heuristic and subjective judgments. A general weakness of the manual techniques is that correct usage often involves tedious and repetitious activities. As the size of the application increases, the tendency is to compromise on the thorough application of the technique, which results in an increasing chance of error.

Automated static analysis tools most often operate on program source code. Two kinds of static analysis can be identified. The first gathers and reports information about a program. Generally, this kind of analysis does not search for any particular type of error in a program. A symbol cross-referencer generator and a consistency check with the specifications are examples of this type.

The second kind of analysis detects specific classes of errors or anomalies in a program. Examples of this type are as follows. Error and anomaly detection techniques detect errors using the following– (1) parsers determine the adherence of a program to the language syntax and may include additional local programming conventions and standards such as percentage of comments per lines of code; (2) techniques for analyzing the consistency of actual and formal parameter interfaces (fig. 4.2-2); (3) techniques for comparing all variable references with their declarations to check for consistency; (4) techniques for analyzing a program for erroneous sequences of events or operations, such as reading from a file before it is opened and using a variable before it is initialized; (5) techniques that determine whether variables in an expression are commensurate (i.e., adding gallons and miles).

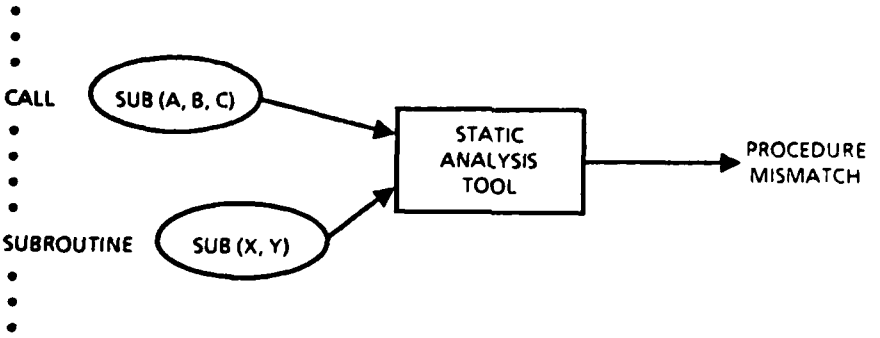


Figure 4.2-2. Module Interface Consistency Check

Structure analysis techniques detect improper subprogram usage and violation of control flow standards. This information is usually given in the form of error reports and a program call graph. In general, various types of status reports are generated as a by-product of static analysis. The section on documentation (4.3.1.3.2) provides useful information on various types of reports.

Program quality analysis techniques provide information such as, a measure for overall program length, the potential smallest volume of an algorithm (or the most succinct form

in which an algorithm could be expressed), a measure of the complexity of the program code, and measures of software qualities such as, reliability, and maintainability.

Input space partitioning techniques are methods of generating test data from the analysis of the input space (all possible input values) and the predicates (condition statements, i.e., IF-THEN-ELSE, WHILE-DO) of a program. The types of errors the test data are sensitive to are errors that occur on or near the boundary of a path, computation errors, wrong path and missing path errors.

Data-flow guided testing techniques have wide applicability in compiler design and optimization activities. Typical data-flow guided testing problems include available expressions, live variables, reaching definitions, and very busy variables. See section 4.3.1.6 for a more in-depth discussion of these terms.

4.2.2 Dynamic Analysis

In contrast to static analysis, which does not involve the execution of the program, dynamic analysis involves actual execution of the program. The principal applications of dynamic analysis includes program testing, debugging, and performance measurement (fig. 4.2-3). This involves the processes of—

a. Preparing for test execution.
b. Test execution.
c. Analysis of test results.

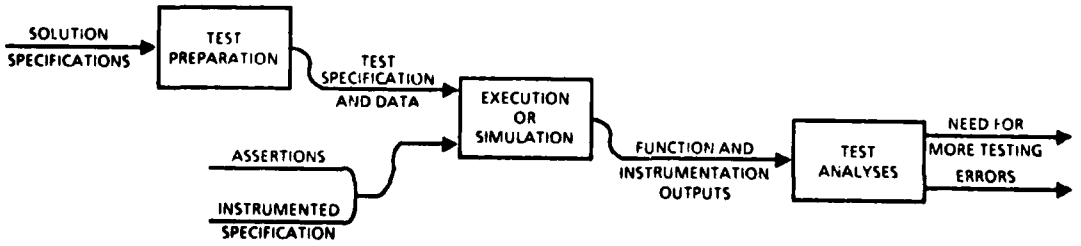


Figure 4.2-3. General Form of Dynamic Analysis

4.2.2.1 Test Preparation

Test preparation is accomplished primarily through manual methods. These method: include a definition of specification-based functional tests and cause-effect graphing Specification-based functional testing is a method of developing test scenarios, test data and expected results through examining the program specifications (in particular, the requirements, design, and program code). Test scenarios based on the requirements have the objective of demonstrating that the functions, performance and interface requirements, and solution constraints are satisfied. Test cases are determined from the design-to-test functions, structures, algorithms, and other elements of the design. Test data are determined from the program to exercise computational structures implemented within the program code.

Preparing for test execution includes test data preparation and formulation of expected results. Test data preparation formulates test scenarios, test cases, and the data to be input to the program. Test scenarios and test cases are chosen as the result of analyzing the requirements and design specifications and the code itself. The test data should demonstrate and exercise externally visible functions, program structures, data structures, and internal functions. Each test case includes a set of input data and the expected results. The expected results may be expressed in terms of final values and as statements (assertions) about intermediate states of program execution.

4.2.2.2 Test Execution

Test execution involves executing a program with prepared test cases and then collecting the results. Testing may be planned and performed in a top-down or bottom-up fashion, or a combination of the two. Top-down testing is performed in parallel with top-down construction in that a module is developed and tested while submodules are left incomplete as stubs or dummy routines. Bottom-up testing consists of testing pieces of code, individual modules, and small collections of modules in that order, before they are integrated into the total program. Bottom-up testing may require the use of test driver or test harness routines as test support tools.

4.2.2.3 Test Analyses

Test coverage analysis captures and reports execution details (e.g., statement or branch execution counts). This dynamic analysis also includes determining the thoroughness of the testing. The process of analyzing the test results involves comparing the actual to expected results. This analysis requires a specification of the expected results for each case. Comparison of actual and expected results may be performed manually, or if the data are machine readable, then an automated comparator may be used. The detection of assertion violations is normally accomplished through analyzing the assertion results generated by the instrumented program.

4.2.2.4 Dynamic Analysis Techniques

The following paragraphs contain summary descriptions of the dynamic analysis testing techniques used in this guidebook. Assertions and cause-effect graphing can be used to assist test preparation.

Development of assertions takes place during the design and programming subphases of the life cycle. In general, assertions are statements that specify the intent of a program's behavioral properties and constraints. Assertions may be generated concerning inputs, outputs, and intermediate steps of each function. A special notation (an assertion language) often is used to specify the assertions. Assertions are developed and inserted into the actual design specification and program code, usually as specially formatted comments.

Instrumentation-based testing techniques are commonly used to assist in test execution. Instrumentation, that is, the insertion of code into a program to measure program characteristics, involves path and structural analysis, performance measurement, assertion checking, and debug aids.

Cause-effect graphing is a technique used to develop test cases based on inputs and input conditions. For each case, the expected outputs are identified. This technique utilizes the requirements and design specifications of a program to develop test cases.

Path and structural analysis techniques provide analytic information concerning the execution of a segment of a program. The number of times a segment is executed is reported. Performance measurement techniques examine the execution of a program in order to provide summary reports on the resource usage of the program. Assertion checking techniques allow the programmer to specify assertion statements that are inserted in to the code. If the assertion is found to be false during execution, an error message is reported. Debug aids are techniques used to control and/or analyze the dynamics of a program during execution. Various commands can be made during execution that prove to be helpful debugging aids. A TRACE command will display control flow information, a DUMP command will display contents of specific storage cells, and a BREAK command will suspend program execution at a specified point in the code.

The following four dynamic analysis techniques can be characterized by unique methodologies: Random testing is a method of detecting computational and control flow errors by executing randomly generated inputs and examining the outputs for correctness. This black-box technique is effective with certain limitations. It is difficult to determine how long a program should be subjected to random inputs to ensure adequate error detection. A program that is only subjected to 1 hour of randomly generated inputs may not be as tested as a program that is subjected to 6 hours of random input. This technique is feasible if computer resources are available to facilitate adequate testing.

Functional testing is a technique that uses the requirements for software product as the primary tool for designing the tests. It is by far the most important and widely used single testing approach.

Mutation testing is a test technique that involves modifying actual statements of the program. Mutation analysis and error seeding are two examples of this technique. Mutation analysis is a method of detecting mutation errors (i.e., those involving alteration, interchanging, or omission of operators, and variables) and also reveals information concerning the thoroughness with which a program has been tested. This technique can be an effective method for detecting errors if an experienced person and automated tools are used in combination. It is necessary for the person to examine the mutant programs that behave identically to the original program to determine whether the mutant is equivalent to the original program or whether the collection of test data

sets is inadequate. Automated tools are essential in expediting the processing of the many mutant programs. At the end of a successful mutation analysis, many errors will be uncovered and the collection of test data sets will be very thorough.

Real-time testing is used to simulate the environment surrounding a system when testing in a live environment is impractical or costly. This technique is mainly used during the system test phase for usually one or more of the following three reasons.

a. To simulate stress and volume tests (e.g., simulating the actions of 100 terminal users on a timesharing system).

b. To create test conditions that are difficult or impossible to create (e.g., for certain hardware failures).

c. When testing in the actual environment is impossible (e.g., testing nuclear-reactor control programs, aerospace systems).This type of testing is effective in detecting errors that other test methods would fail to uncover.

4.2.3 Symbolic Testing

Symbolic execution is method of interpreting programs by deriving mathematical expressions for the values of variables rather than actually computing their numerical values. The expressions produced show a perspective of the progress of computations which is very different from other means of testing such as tracing intermediate values. The additional information obtained from symbolic execution has been shown to improve detection of several kinds of program errors (HOW77).

4.2.4 Formal Analysis

Formal analysis is a more rigorous approach to software testing which involves proving properties of computation performed by programs. This technique provides probably the highest degree of assurance of program correctness but is also the most difficult to apply. Two prerequisites for formal analysis of a program are precise specifications of the inputs and required outputs for the computation, and formal specifications of the semantics of the programming language used. Program correctness is proven by showing that, given the specified input conditions and the rules of the programming language, execution of the program will terminate and produce the desired output conditions.

A type of formal analysis is proof of program correctness. There are two basic approaches used with this technique. One approach is to prove the correctness of a whole program. The second approach is to prove the correctness of particular program properties.

The basic strategy to both approaches is to construct a set of reasoning to show that a solution specification satisfies its requirement(s). Typically, this is done by comparing the inferred transformations to the functional transformations dictated by the specifications of intent (fig. 4.2-4).

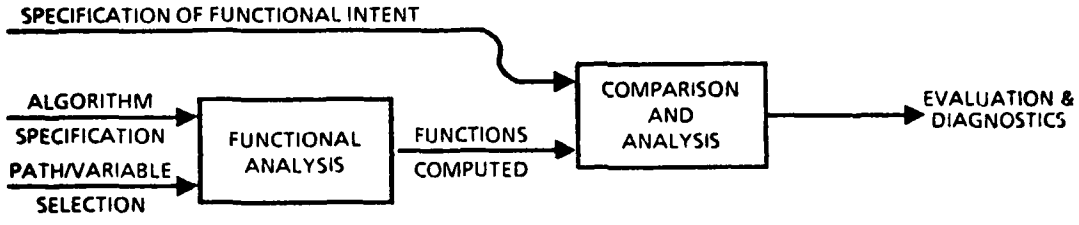


Figure 4.2-4. General Form of a Formal Functional Analysis

To do the comparison, control paths are selected through the algorithmic specification. For each path the values of each data object encountered are computed. Each value is not computed as a number but rather as a function, or formula. Input are not assigned specific values but rather are treated as free or unbound variables. As a result, at the end of tracing down the selected path, the functional relationships of outputs to inputs is determined. These functions or formulas are then compared to specifications of functional intent to see if the solution specification is correct as a value transformer.

Proof of correctness is limited by its strict dependence on the validity of the assumptions on which the analysis is based. Proof of correctness has not yet reached its full potential as a testing technique for several reasons. For one, program specifications are rarely written with sufficient precision to permit a rigorous comparison of intent with the implemented program. Also, accomplishing the analysis manually is extremely tedious and difficult (thus, prone to error).

4.3 DETAILED TECHNIQUE DESCRIPTIONS AND CHARACTERISTICS

Each technique description is presented in a standard format as described in the following sections.

a. Information Input. Contains the input information required to use this technique.

b. Information Output. Contains a description of the results of the technique.

c. Outline of Method. Contains a basic outline of the methodology. In some cases, it is beyond the scope of this handbook to outline the methodology fully; a reference is provided in such cases.

d. Example. Provides an example of the technique if it is possible. Again, an example of some techniques would be lengthy and thus beyond the scope of this handbook. References are provided. The use of specific tools in the text are for examples only; those tools are not being promoted for use.

e. Effectiveness. Provides profile information, such as follows:

1. The types of errors detected.
2. The degree to which the technique detects those errors based on the usage of the technique.

f. Applicability. Provides information dealing with the following usages.

1. During which software development test phase(s) as defined in figure 5.1-1.
2. To which specific types of software applications
   (a) Scientific.
   (b) Complex control flow.
   (c) Large versus small applications.

g. Maturity. This section provides information on the current status of a technique with respect to usage.

1. Still in developmental and experimental stage.
2. Has been tested and widely used.

h. User Training. This section contains an estimate of the learning time and training needed to successfully implement the technique.

i. Costs. This section identifies the necessary resources required to use the technique. (e.g., human effort, computer time, or any type of overhead).

j. References. References are provided at the end of each technique description. The complete reference can be found in the bibliography (sec. 4.6).

4.3.1 Static Analysis Techniques

4.3.1.1 Code Reviews and Walkthroughs

This section contains descriptions of peer reviews, formal reviews, and associated walkthroughs. These test techniques involve the reading or visual inspection of a program by a team of people. The objective of the team is to find errors, but not to find solutions to the errors.

4.3.1.1.1 Peer Review

A peer review is a process by which project personnel perform a detailed study and evaluation of code, documentation, or specification. The team peer review refers to product evaluations conducted by individuals of equal rank, responsibility, or of similar experience and skill. There are a number of review techniques that fall into the overall category of a peer review. Code reading, round-robin reviews, walkthroughs, and inspections are examples of peer reviews that differ in formality, participant roles and responsibilities, output produced, and input required.

a. Information Input. The input to a particular peer review will vary slightly depending on which form of peer review is being conducted. In general, each of the forms of peer review require that some sort of review package is assembled and distributed. This package commonly contains a summary of the requirement(s) that are the basis for the product being reviewed. Other common inputs are differentiated by the stage of the software life cycle currently in process. For example, input to a peer review during the coding phase would consist of program listings, design specifications, programming

standards and a summary of results from the design peer review previously held on the same product. Common input to particular forms of peer review are described as follows:

- Code-reading review—
  - Component requirements.
  - Design specifications.
  - Program listings.
  - Programming standards.
- Round-robin review—
  - Component requirements.
  - Design or code specifications.
  - Program listings (if during coding phase).
- Walkthrough—
  - Component requirements.
  - Design or code specifications.
  - Program listings (if coding phase walkthrough).
  - Product standards.
  - Backup documentation (i.e., flowcharts, HIPO charts, data dictionaries).
  - Question list (derived by participants prior to review).
- Inspection—
  - Component requirements.
  - Design or code specifications.
  - Program listings (if during coding phase).
  - Product standards.
  - Backup documentation.
  - Checklist (containing descriptions of particular features to be evaluated).

b. Information Output. The output from a peer review varies by form of review. One output common to each form of a peer review is a decision or concensus about the product under review. This is usually in the form of a group approval of the product as is, an approval with recommended modifications, or a rejection (and rescheduled review date.) Specific outputs by form of peer review are as follows.

- Code-reading and round-robin reviews—
  - Informal documentation of detected problems.

- Recommendation to accept or reject reviewed product.
- Dependency list (containing the relationship of coding constructs with variables).

- Walkthrough—
  - Action list (formal documentation of problems).
  - Walkthrough form (containing review summary and group decision).
- Inspection—
  - Inspection schedule and memo (defining individual roles and responsibilities, inspection agenda and schedule).
  - Problem definition sheet.
  - Summary report (documenting error correction status and related statistics on the errors).
  - Management report (describing errors, problems, component status to management).

c. Outline of Method. The peer review methodology and participant responsibilities vary by form of review. Summaries of these methodologies are provided in the latter part of this section. However, there are a few features common to each methodology, which are described in the following paragraphs.

Most peer reviews are not attended by management. (An exception is made in circumstances where the project manager is also a designer, coder, or tester, usually on very small projects.) The presence of management tends to inhibit participants, since they feel that they are personally being evaluated. This would be contrary to the intent of peer reviews, that of studying the product itself.

The assembly and distribution of project review materials prior to the conduct of the peer review is another common feature. This allows participants to spend some amount of time reviewing the data to become better prepared for the review.

At the end of most peer reviews, the group arrives at a decision about the status of the review product. This decision is usually communicated to management.

Most reviews are conducted in a group organization as opposed to individually by participants or by the project team itself. While this may seem an obvious feature, it

bears some discussion. Most organizations doing software development and/or maintenance employ some variation of a team approach. Some team organizations are described as follows:

- Conventional team. A senior programmer directs the efforts of one or more less experienced programmers.
- Egoless team. Programmers that are of about equal experience share product responsibilities.
- Chief programmer team. A highly qualified senior programmer leads the efforts of other team members for which specific roles and responsibilities have been assigned (i.e., backup programmer, secretary, librarian).

The group participating in the peer review is not necessarily the same as the team organized to manage and complete the software product. The review group is likely to be composed of a subset of the project team plus other individuals as required by the form of review being held and the stage of the life cycle in process. The benefits of peer reviews are unlikely to be attained if the group acts separately, without some designated responsibilities. Some roles commonly used in review groups are described below. These roles are not all employed in any one review.

- Group and review leader. The individual designated by management with planning, directing, organizing and coordinating responsibilities. Usually has responsibilities after the review to ensure that recommendations are implemented.
- Designer. The individual responsible for the translation of the product into a plan for its implementation.
- Implementer. The individual responsible for developing the product according to the plan detailed by the designer.
- Tester. The individual responsible for testing the product as developed by the implementer.
- Coordinator. The individual designated with planning, directing, organizing and coordinating responsibilities.
- Producer. The individual whose product is under review.
- Recorder. The individual responsible for documenting the review activities during the review.
- User representative. The individual responsible for ensuring that the user's requirements are addressed.

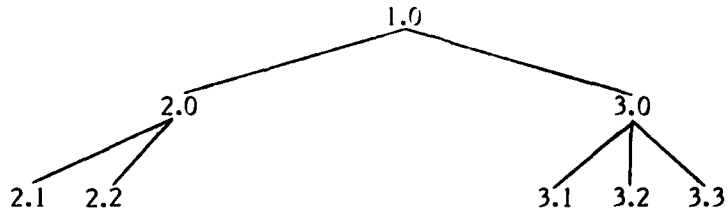
- Standards representative. The individual responsible for ensuring that product standards are conformed to.
- Maintenance representative. The individual who will be responsible for updates or corrections to the installed product.
- Others. Individuals with specialized skills or responsibilities which acquire their contributions during the peer review.

While the forms of peer reviews have some similarities and generally involve designation of participant roles and responsibilities, they are different in application. The remainder of this section will summarize the application methods associated with the forms of peer reviews previously introduced.

Code-Reading Review. Code reading is a line-by-line study and evaluation of program source code. It is generally performed on source code which has been compiled and is free of syntax errors. However, some organizations practice code-reading on uncompiled source listings or hand-written code on coding sheets in order to remove syntax and logic errors prior to code entry. Code reading is commonly practiced on top-down, structured code and becomes cost ineffective when performed on unstructured code.

The optimum size of the code-reading review team is 3-4. The producer sets up the review and is responsible for team leadership. Two or three programmer/analysts are selected by the producer based upon their experience, responsibilities with interfacing programs, or other specialized skill.

The producer distributes the review input (see input section) about 2 days in advance. During the review the producer and the reviewers go through each line of code checking for features which will make the program more readable, usable, reliable and maintainable. Two types of code-reading may be performed: reading for understanding and reading for verification. Reading for understanding is performed when the reader desires an overall appreciation of how the program module works, its structure, what functions it performs, and whether it follows established standards. Assuming that the following figure depicts the structure of a program component, a reviewer reading for understanding would review the modules in the the following order: 1.0, 2.0, 2.1, 2.2, 3.0, 3.1, 3.2, 3.3.



In contrast to this top-to-bottom approach, reading for verification implies a bottom-up review of the code. The component depicted above would be perused in the following order: 3.3, 3.2, 3.1, 3.0, 2.2, 2.1, 2.0, 1.0. In this manner it is possible to produce a dependency list detailing parameters, control switches, table pointers, and internal and external variables used by the component. The list can then be used to ensure hierarchical consistency, data availability, variable initiation, etc. Reviewers point out any problems or errors detected while reading for understanding or verification during the review.

The team then makes an informal decision about the acceptability of the code product and may recommend changes. The producer notes suggested modifications and is responsible for all changes to the source code. Suggested changes are evaluated by the producer and need not be implemented if the producer determines that they are invalid.

There is no mechanism to ensure that change is implemented or to follow-up on the review.

Round-Robin Review. A round-robin review is a peer review where each participant is given an equal and similar share of the product being reviewed to study, present and lead in its evaluation.

A round-robin review can be given during any phase of the product life cycle and is also useful for documentation review. In addition, there are variations of the round-robin review which incorporate some of the best features from other peer review forms but continue to use the alternating review leader approach. For example, during a round-robin inspection, each item on the inspection checklist is made the responsibility of alternating participants.

The common number of people involved in this type of peer review is 4-6. The meeting is scheduled by the producer, who also distributes some high-level documentation as input. The producer will either be the first review leader or will assign this responsibility to another participant. The temporary leader will guide the other participants (who may be implementors, designers, testers, users, maintenance representatives, etc.) through the first unit of work. This unit may be a module, paragraph, line of code, inspection item, or other unit of manageable size. All participants (including the leader) have the opportunity to comment on the unit before the next leader begins the evaluation of the next unit. The leaders are responsible for noting major comments raised about their piece of work. At the end of the review all the major comments are summarized and the group decides whether or not to approve the product. No formal mechanism for review follow-up is used.

Walkthroughs. This type of peer review is more formal than the code-reading review or round-robin review. Distinct roles and responsibilities are assigned prior to review. Pre-review preparation is greater and a more formal approach to problem documentation is stressed. Another key feature of this review is that it is presented by the producer. The most common walkthroughs are those held during design and code yet recently they are being applied to specifications documentation and test results.

The producer schedules the review and assembles and distributes input. In most cases the producer selects the walkthrough participants (although sometimes this is done by management) and notifies them of their roles and responsibilities. The walkthrough is usually conducted with less than seven participants and lasts not more than 2 hours. If more time is needed a break must be given or the product should be reduced in size. Roles usually included in a walkthrough are producer, coordinator, recorder, and representatives of user, maintenance and standards organizations.

The review is opened by the coordinator, yet the producer is responsible for leading the group through the product. In the case of design and code walkthroughs, the producer simulates the operation of the component, allowing each participant to comment based on his area of specialization. A list of problems is kept and at the end of the review, each participant signs the list or other walkthrough form indicating whether the product is accepted as-is, accepted with recommended changes, or rejected. Suggested changes are made at the discretion of the producer. There are no formal means of follow-up on the

review comments. However, if the walkthrough review is used for products throughout the life cycle (i.e., specification, design, code and test walkthrough), comments from past reviews can be discussed at the start of the next review.

Inspections. Inspections are the most formal, commonly-used form of peer review. The key feature of an inspection is that it is driven by the use of checklists to facilitate error detection. These checklists are updated as statistics indicate that certain types of errors are occurring more or less frequently than in the past. The most commonly held types of inspections are conducted on the product design and code, although inspections may be used during any life cycle phase. Inspections should be short since they are often quite intensive. This means that the product component to be reviewed must be of small size. Specifications or design which will result in 50-100 lines of code are normally manageable. This translates into an inspection of 15 minutes to 1 hour, although complex components may require as much as 2 hours. In any event, inspections of more than 2 hours are generally less effective and should be avoided.

Two or three days prior to the inspection the producer assembles the input to the inspection and gives it to the coordinator for distribution. Participants are expected to study and make comments on the materials prior to the review.

The review is lead by a participant other than the producer. Generally, the individual who will have the greatest involvement in the next phase of the product life cycle is designated as reader. For example, a requirements inspection would likely be lead by a designer, a design review by an implementer, and so forth. The exception to this occurs for a code inspection which is lead by the designer. The inspection is organized and coordinated by an individual designated as the group leader or coordinater.

The reader goes through the product component, using the checklist as a means to identify common types of errors as well as standards violations. A primary goal of an inspection is to identify items which can be modified to make the component more understandable, maintainable, or usable. Participants (identified earlier in this section) discuss any issues which they identified in pre-inspection study.

At the end of the inspection an accept/reject decision is made by the group and the coordinator summarizes all the errors and problems detected and provides this list to all

participants. The individual whose work was under review (designer, implementer, tester) uses the list to make revisions to the component. When revisions are implemented, the coordinator and producer go through a mini-review using the problem list as a checklist.

The coordinator then completes management and summary reports. The summary report is used to update checklists for subsequent inspections.

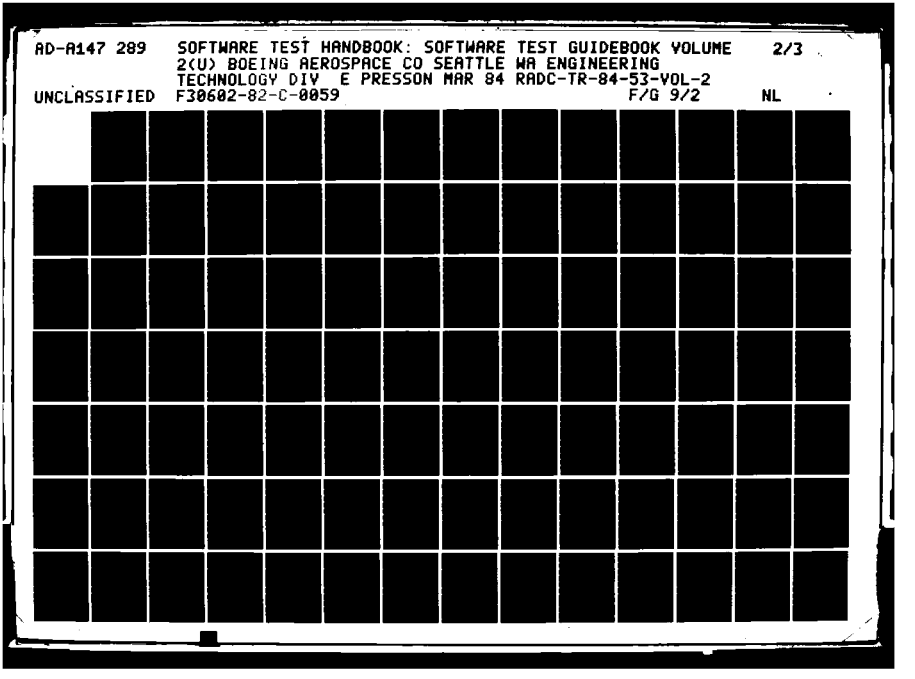
d. Example. This section contains an example describing a code-reading review. Three days prior to estimated completion of coding, the producer of a program component begins preparation for a code-reading review. The component is composed of 90 lines of Fortran code and associated comments. The producer obtains copies of the source listing and requirements and design specifications for the component and distributes them to three peers, notifying them of the review date and place.

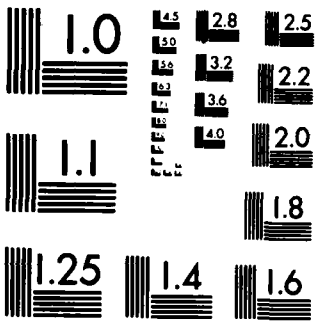
Each reviewer reads the code for general understanding; reviewing a major function and its supporting functions prior to reviewing the next major function.

One reviewer notes an exception to the programming standards. Another thinks that the data names are not meaningful. The third has found several comments which inaccurately represent the function they describe. Each reviewer makes a note of these points as well as any comments about the structure of the component. Next, the requirements are studied to ensure that each requirement is addressed by the component. It appears that the requirements have all been met.

The code-reading review is led by the producer. After a brief description of the component and its interfaces, the producer leads the reviewers through the code. Rather than progressing through the component from top to bottom, the decision is made to perform code-reading from the bottom up. This form of code-reading is used to verify the components correctness.

As the code is being perused one of the reviewers is made responsible for keeping a dependency list. As each variable is defined, referenced, or modified. a notation is made on the list.


AD-A147 289 SOFTWARE TEST HANDBOOK: SOFTWARE TEST GUIDEBOOK VOLUME 2/3
2(U) BOEING AEROSPACE CO SEATTLE WA ENGINEERING
TECHNOLOGY DIV E PRESSON MAR 84 RADC-TR-84-53-VOL-2
UNCLASSIFIED F30602-82-C-0059 F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

The verification code reading uncovers the use of a variable prior to its definition. This error is documented on an error list by the producer. In addition, each of the problems detected earlier during the code-reading (as performed by each individual) is discussed and documented.

At the end of the review, the error list is summarized to the group by the producer. Since no major problems were detected, the participants agree to accept the code with the agreed-to minor modifications. The producer then uses the error/problem list for reference when making modifications to the component.

e. Effectiveness. Studies have been conducted to identify the following qualitative benefits by forms of peer reviews.

- Higher status visibility.
- Decreased debugging time.
- Early detection of design and analysis errors which would be much more costly to correct in later development phases.
- Identification of design or code inefficiencies.
- Ensuring adherence to standards.
- Increased program readability.
- Increased user satisfaction.
- Communication of new ideas or technology.
- Increased maintainability.

Little data are available which identifies the quantitative benefits attributable to the use of a particular form of peer review. However, one source estimates that the number of errors in production programs was reduced by a factor of ten by utilizing walkthroughs. Computational and logic errors are those types of errors detected by this technique. Another source estimates that a project employing inspections achieved 23% higher programmer productivity than with walkthroughs. No data was available indicating the amount of increased programmer productivity attributable to the inspections alone.

f. Applicability. Peer reviews are applicable to large or small projects during design verification, unit test, and module test phases and are not limited by project type or complexity.

g. Maturity. Peer reviews are widely used and have demonstrated to be quite effective in detecting errors despite the informality of the methodology. This technique has been developed for quite sometime.

h. User Training. None of the peer reviews discussed require extensive training to implement. They do require familiarity with the concept and methodology involved. Experience has shown that peer reviews are most successful when the individual with responsibility for directing the review is knowledgable about the process and its intended results.

i. Costs. The reviews require no special tools or equipment. The main cost involved is that of human resources. If the reviews are conducted in accordance with the resource guidelines expressed in most references, the costs of peer reviews should be negligible compared with the expected returns. Most references suggest that peer reviews should be no longer than 2 hours, preferably 1/2 to 1 hour. Preparation time can amount to as little as 1/2 hour and should not require longer than 1/2 day per review.

j. References.

(COD 76)
(DAL 77)
(FAG 76)
(FRE 77)
(GLA 78)
(SHN 80)
(SYS 77)
(YOU 77)
(MYE 78)

4.3.1.1.2 Formal Review

Formal reviews constitute a series of reviews of a software system, usually conducted at major milestones in the software development life cycle. They are used to improve development visibility and product quality and provide the basic means of communication between the project team, company management, and user representatives. Formal reviews are most often implemented for medium- to large-size development projects, although small projects employ a less rigorous form of the technique.

The most common types of formal reviews are held at the completion of the Requirements, Preliminary Design, Detailed (Critical) Design, Coding, and Installation phases. While names of these reviews may vary by company, some generally recognized names are Requirements Review, Preliminary Design Review (PDR), Critical Design Review (CDR), Code Construction Review, and Acceptance Test Review.

a. Information Input. The input to a particular formal review will vary slightly depending on the stage of the life cycle just completed. In general, each formal review will require that some sort of review package be assembled and then distributed at a review meeting. This package commonly contains a summary of the requirements that are the basis for the product being reviewed. These and other common inputs to formal reviews fall into three main categories, described as follows.

Project documents. These are documents produced by the development team to describe the system. The specific documents required are dependent upon the life cycle phase just completed. For example: a review conducted at the conclusion of the requirements phase would necessitate availability of Functional Specifications or System/Subsystem Specifications.

Backup documentation. This type of input is documentation which is not usually contractually required, yet preparation of which is necessary to support systems development or otherwise record project progress. Specific types of backup documentation vary by the phase for which the review is performed. Rough drafts of user and maintenance manuals are examples of backup documentation examined during a design review to plan for continuation of the project. Program listings are an example of backup documentation utilized during a code construction review.

Other inputs. All other inputs are primarily used to clarify or expand upon the project documents and backup documents. They may include viewfoils and slides prepared by project management for the formal review meeting, the minutes of the previous phase review meeting, or preliminary evaluations of the project documents under review.

b. Information Output. The information output associated with a formal review generally falls into the following categories.

Management reports. These are written reports from the project manager to upper management describing the results of the review, problems revealed, proposed solutions, and any upper management assistance required.

Outside reviewer reports. These are written reports to the project manager from participants of the review who have not worked on the project. These reports provide

outside reviewers an opportunity to express their appraisal of the project status and the likelihood of meeting project objectives. It also allows them to make suggestions for correcting any deficiencies noted.

Action items. This is a list of all required post-review action items to be completed before a review can be satisfactorily closed out. With each item is an indication of whether customer or contractor action is required for resolution.

Review minutes. This is a written record of the review meeting proceedings which are recorded by a designee of the leader of the review team. The minutes of the review are distributed to each review team member after the completion of the review meeting.

Decision to authorize next phase. A decision must be reached at any formal review to authorize initiation of the next life cycle phase.

Understanding of project status. At the conclusion of any formal review there should be a common understanding of project status among the project personnel present at the review.

c. Outline of Method. The methodology of formal reviews is outlined as follows.

Participants. The participants in a formal review are often selected from the following group of people:

- Project manager.
- Project technical lead.
- Other project team members—analysts, designers, programmers.
- Client.
- User representative(s).
- Line management of project manager.
- Outside reviewers—quality assurance personnel, experienced people on other projects.
- Functional support personnel—finance, technology personnel.
- Subcontractor management, if applicable.
- Others—configuration management representative, maintenance representative.

The process. Formal reviews should be scheduled and organized by project management. Each review must be scheduled at a meaningful point during software development. The review effectively serves as the phase milestone for any particular phase. There are five basic steps involved in every formal review.

Preparation. All documentation that serves as source material for the review must be prepared prior to the review. These materials may be distributed to each review participant before the review meeting in order to allow reviewers sufficient time to review and make appraisals of the materials. The location and time of the review meeting must be established, participants must be identified, and an agenda planned.

Overview presentation. At the review meeting, all applicable product and backup documentation is distributed and a high-level summary of the product to be reviewed is presented. Objectives of the review are also given.

Detailed presentation. A detailed description of the project status and progress achieved during the review period is presented. Problems are identified and openly discussed by the review team.

Summary. A summary of the results of the review is given. A decision about the status of the review is made. A list of new action items is constructed and responsibility for completion of each item is assigned.

Followup. The completion of all action items is verified. All reports are completed and distributed.

d. Example. Two weeks prior to estimated completion of the requirements document, the producer of a program receives notification from his client that a requirements review is desired. The client proceeds in selecting a chairman to conduct the review. As participants in the review, the chairman selects the project manager, project technical lead, a member of the requirements definition team, and a member of the requirements analysis team. The client also has indicated that he would like to include the following people in the review: a representative from the user shop, a reviewer from an independent computing organization, and a representative from his own organization.

The chairman informs all review participants of the date, time, and location of the review. Ten days prior to the meeting, the chairman distributes all documents produced by the requirements definition and analysis teams (requirements document, preliminary plans, other review material) to each review participant. In preparation of the meeting, each reviewer critically reviews the documents. The user representative is puzzled over the inclusion of a requirement concerning the use of a proposed database. The reviewer from the outside computing organization notes that the version of the operating system to be used in developing the software is very outdated. A representative of the client organization has a question concerning the use of a subcontractor in one phase of the project. Each reviewer submits his comments to the chairman before the scheduled review meeting. The chairman receives the comments and directs each to the appropriate requirements team member to allow proper time for responses to be prepared.

The requirements review meeting begins with a brief introduction by the chairman. All participants are introduced, review materials are listed, and the procedure for conducting the review is presented. A presentation is then given summarizing the problem that led to the requirements and the procedure that was used to define these requirements. At this time, the user representative inquires about the requirement concerning the use of a particular database as stated in the requirements document. The project technical lead responds to this question. The user representative accepts this response, which is so noted by the recorder in the official minutes of the meeting.

The meeting continues with an analysis of the requirements and a description of the contractor's planned approach for developing a solution to the problem. At this time, the questions from the client representative and the outside computing organization are discussed. The project manager responds to questions concerning the use of a subcontractor on the project. Certain suggestions have been made which require the approval of the subcontractor. These suggestions are placed on the action list. The technical lead acknowledges the problems that the independent computing organization has pointed out. He notes that certain software vendors must be contacted to resolve the problem. This item is also placed on the action list. A general discussion among all review team members follows.

At the end of the review, the chairman seeks a decision from the reviewers about the acceptability of the requirements document. They agree to give their approval, providing

that the suggestions noted on the action list are thoroughly investigated. All participants agree to this decision and the meeting is adjourned.

The chairman distributes a copy of the minutes of the meeting, including action items, to all participants. The project manager informs the subcontractor of the suggestions made at the meeting. The subcontractor subsequently agrees with the suggestions. The project technical lead contacts the software vendor from which the current operating system was purchased and learns that the latest version can be easily installed before it is needed for this project. He notifies the project manager of this, who subsequently approves its purchase. The requirements document is appropriately revised to reflect the completion of these action items. The chairman verifies that all action items have been completed. The project manager submits a management report to management, summarizing the review.

e. Effectiveness. Since the cost to correct an error increases rapidly as the development process progresses, detection of errors by the use of formal reviews is an attractive prospect. Computational and logic errors are the types of errors detected by this technique.

Some of the qualitative benefits attributable to the use of formal reviews are given below:

- Highly visible systems development.
- Early detection of design and analysis errors.
- More reliable estimating and scheduling.
- Increased product reliability, maintainability.
- Increased education and experience of all individuals involved in the process.
- Increased adherence to standards.
- Increased user satisfaction.

Little data are available which identifies the quantitative benefits attributable to the use of formal reviews.

Experience with this technique indicates that it is most effective on large projects. The costs involved in performing formal reviews on small projects, however, may be sufficiently large enough to consider lessening the formality of the reviews or even eliminating or combining some of them.

f. Applicability. Formal reviews are applicable to large or small projects, during design verification and unit test phases and are not limited by project type or complexity.

g. Maturity. This is a widely used technique.

h. User Training. This technique does not require any special training. However, the success or failure of a formal review is dependent on the people who attend. They must be intelligent, skilled, knowledgeable in a specific problem area, and be able to interact effectively with other team members. The experience and expertise of the individual responsible for directing the review is also critical to the success of the effort.

i. Costs. The method requires no special tools or equipment. The main cost involved is that of human resources. If formal reviews are conducted in accordance with the resource guidelines expressed in most references, the costs of reviews are negligible compared with the expected returns. Most references suggest that formal review meetings should not require more than 1 to 2 hours. Preparation time can amount to as little as 1/2 hour and should not require longer than 1/2 day per review.

j. References.

(FRE 77)
(GLA 79A)
(MEY 75)
(SHN 80)
(WEI 71)

4.3.1.2 Error and Anomaly Detection Techniques

This section contains techniques that detect errors without executing the program (statically). Testing techniques that deal with syntax checking (uninitialized variables, unreachable code), coding standards auditing, and data checking (set-use, data flow, units consistency) are described in this section.

4.3.1.2.1 Code Auditing

Code auditing involves examining the source code and determining whether prescribed programming standards and practices have been followed.

a. Information Input. The information input to code auditing is the source code to be analyzed.

b. Information Output. The information output by code auditing is a determination of whether the code under analysis adheres to prescribed programming standards. If errors exist, information is generated detailing which standards have been violated and where the violations occur. This information can appear as error messages included with a source listing or as a separate report. Other diagnostic information, such as a cross-reference listing, may also be output as an aid to making error corrections.

c. Outline of Method. Code auditing provides an objective, reliable means of verifying that a program complies with a specified set of coding standards. Some common programming conventions that code auditing can check for are as follows.

Correct syntax. Do all program statements conform to the specifications of the language definition?
Portability. Is the code written so that it can easily operate on different computer configurations?

Use of structured programming constructs. Does the code make proper use of a specified set of coding constructs such as IF-THEN-ELSE or DO-WHILE?

Size. Is the length of any program unit not more than a specified number of statements?

Commentary. Is each program unit appropriately documented? For example, is each unit preceded by a block of comments that indicate the function of the unit and the function of each variable used?

Naming conventions. Do the names of all variables, routines, and other symbolic entities follow prescribed naming conventions?

Statement labeling. Does the numeric labeling of statements follow an ascending sequence throughout each program unit?

Statement ordering. Do all statements appear in a prescribed order? For example, in a Fortran program, do all FORMAT statements appear at the end and data specification statements before the first executable statement of a routine?

Statement format. Do all statements follow a prescribed set of formatting rules that improve program clarity? For example, are all DO-WHILE loops appropriately indented?

As demonstrated by this list, code auditing may vary in sophistication according to their function. Each form, however, requires some form of syntax analysis to be performed. Code must be parsed and given an internal representation suitable for analysis. Because this type of processing is found in many static analysis tools, code auditing may be part of a more general tool having many capabilities. For example, a compiler is a form of code auditing that checks for adherence to the specifications of a language definition. PFORT, a tool used to check Fortran programs for adherence to a portable subset of American National Standard (ANS) Fortran, also has the capability of generating a cross-reference listing.

Code auditing is useful to programmers as a means of self-checking their routines prior to turnover for integration testing. These tools are also of value to software product assurance personnel during integration testing, prior to formal validation testing, and again prior to customer delivery.

d. Example. An example of code auditing follows:

Application. A flight control program is to be coded entirely in PFORT, a portable subset of ANS Fortran. The program is to be delivered to a military government agency who will install the software on various computer installations. In addition, the customer requires that each routine in the program be clearly documented in a prescribed format. All internal program comments are to be later compiled as a separate source of documentation for the program.

Error. A named common block occurs in several routines in the program. In one routine, the definition of a variable in that block has been omitted because the variable is not referenced in that routine. This is a violation of a rule defined in PFORT, however, which requires that the total length of a named common block agree in all occurrences of that block.

Error discovery. A code auditor which checks Fortran for adherence to PFORT detects this error immediately. The programmer of this routine is informed that the routine is to be appropriately modified and that any confusion over the use of the variable is to be clarified in the block of comments that describe the function of each defined variable in the routine. A code auditor that checks for the presence of appropriate comments in each routine is used to verify that the use of the variable is appropriately documented. At the end of code construction, all such internal program documentation will be collated and summarized by another code auditor which processes machine-readable documentation embedded in source code.

e. Effectiveness. Code auditing is very effective in certifying that software routines have been coded in accordance with prescribed standards. It is much more reliable than manually performed code audits and are highly cost effective as they are less time consuming than manual audits.

f. Applicability. Code auditing can be applied generally to any type of source code and is applicable during design verification through PQT/FQT test phases. However, each specific tool will be language dependent (i.e., will operate correctly only for specified source languages), and will only accept input that appears in a prescribed format.

g. Maturity. Code auditors have been used with favorable results and have been extensively automated. Code auditing is a highly mature technique.

h. User Training. No special training is required to implement this technique. As code auditing may be implemented by a wide variety of people (programmers, managers, quality assurance personnel, customers), ease in their use is an important attribute. In order to use this technique effectively, however, some learning is required to gain familiarity with the standards on which the auditing is based.

i. Costs. Code auditing is generally very inexpensive as their overhead is usually no more than the cost of a compilation.

j. References.

(BRO 78)
(FIS 74)
(RYD 75)
(HEI 82)
(BRO 82)
(SMI 76)

4.3.1.2.2 Interface Checking

Interface checking involves analyzing the consistency and completeness of the information and control flow between components, modules or procedures of a system.

a. Information Input. Information needed to do interface checking consists of either—

- A formal representation of system requirements.
- A formal representation of system design.
- A program coded in a high-level language.

b. Information Output. The output information of this technique shows module interface inconsistencies and errors. The information can be provided as error messages included with a source listing or as a separate report.

c. Outline of Method. Interface checking will analyze a computer processable form of a software system requirements specification, design specification or code. The method for each of the three representations -requirements, design, and code - will be illustrated below by examining the interface checking capabilities of three existing tools. These three tools were chosen primarily for explanatory purposes and should not be regarded as a recommendation.

PSL/PSA (Problem Statement Language/Problem Statement Analyzer) (BOE 75) is an automated requirements specification tool. Basically, PSL/PSA describes system requirements as a system of inputs, processes and outputs. Both information and control flow are represented within PSL. Interface checking performed by PSA consists of ensuring that all data items are used and generated by some process and that all processes use data. Incomplete requirements specification are, therefore, easily detected.

The Design Assertion Consistency Checker (DACC) (PRE 79) is a tool which analyzes module interfaces based on a design which contains information describing for each module the nature of the inputs and outputs. This information is specified using assertions to indicate the number and order of inputs, data types, units (e.g., feet or radian), acceptable ranges, and so on. DACC checks module calls as specified in the design against the assertions for the called module for consistency and produces an inconsistency report where assertions have been violated.

PFORT is a static analysis tool which primarily is used for checking Fortran programs for adherence to a portable subset of the Fortran language but it also performs subprogram interface checking. PFORT checks the matching of actual with dummy arguments checking for unsafe references, such as constraints being passed as arguments, which are subject to change within the called subprogram.

Interface checking capabilities can also be included within a particular language's compiler as well. For example, Ada (TEI 77) provides a parameter passing mechanism whereby parameters are identified to be input or output or input/output. Moreover data type and constraints (e.g., range and precision) must match between the actual arguments and the formal parameters (in non-generic subprograms).

d. Example. An example of interface checking is presented in this section.

Application. A statistical analysis package written in Fortran utilizes a file access system to retrieve records containing data used in the analysis.

Error. The primary record retrieval subroutine is always passed as the last argument in the argument list of the subroutine call. A statement number in the calling program is to receive control in case an abnormal file processing error occurs. One program, however, fails to supply the needed argument. The compiler is not able to detect the error. Moreover, the particular Fortran implementation is such that no execution time error occurs until a return to the unspecified statement number is attempted at which time the system crashes.

Error discovery. This error can easily be detected by using an interface checker at either the design (e.g., DACC) or coding phase (e.g., PFORT) of the software development activity. Both DACC and PFORT can detect incorrect numbers of arguments.

e. Effectiveness. Interface checking is very effective at detecting a class of errors which can be difficult to isolate if left to testing. This technique will generally check for—

- Modules which are used but not defined.
- Modules which are defined but not used.
- Incorrect number of arguments.

- Data type mismatches between actual and formal parameters.
- Data constraint mismatches between actual and formal parameters.
- Data usage anomalies.

They are generally more cost effective if provided as a capability within another tool such as a compiler, data flow analyzer or a requirements/design specification tool.

f. Applicability. Interface checking is applicable during algorithm confirmation through integration test and independent of application type and size.

g. Maturity. This technique is widely used with the aid of automated tools (DACC and PFORT).

h. User Training. The use of this technique requires only a very minimal learning effort.

i. Costs. Interface checking can be quite inexpensive to use, usually much less than the cost of a compilation.

j. References.

(BOE 75)	(HEI 82)	(GAN 80)
(PRE 79)	(TEI 77)	(MEL 79)
(RYD 75)	(BRO 82)	(MEL 81)

4.3.1.2.3 Physical Units Checking

Many (scientific, engineering, and control) programs perform computations whose results are interpreted in terms of physical units, such as feet, meters, watts, and joules. Physical units checking enables specification and checking of units in program computations, in a manner similar to dimensional analysis. Operations between variables which are not commensurate, such as adding gallons and feet, are detected.

a. Information Input. Units checking requires three things to be specified within a program: the set of elementary units used (such as feet, inches, acres), relationships between the elementary units (such as feet = 12 inches, acre = 43,560 $feet^2$), and the association of units with program variables. The programming language used must support such specifications, or the program must be preprocessed by a units checker.

b. Information Output. The information output depends upon the specific capabilities of the language processor or preprocessor. At a minimum, all operations involving variables which are not commensurate are detected and reported. If variables are commensurate, but not identical, (i.e., they are the same type of quantity, such as units of length, but one requires application of a scaler multiplier to place it in the same units as the other) the system may insert the required multiplication into the code, or may only report what factor must be applied by the programmer.

c. Outline of Method. The specification of the input items is the extent of the actions required by the user. Some systems may allow the association of a units expression with an expression within the actual program. Thus one may write LOTSIZE = (LENGTH * WIDTH feet^2) as a boolean expression, where the product of LENGTH and WIDTH must units of square feet. The process of ensuring that LENGTH * WIDTH is in square feet is the responsibility of the processing system.

d. Example. A short program in Pascal-like notation is shown for computing the volume and total surface area of a right circular cylinder. The program requires as input the radius of the circular base and the height of the cylinder. Because of peculiarities in the usage environment of the program, the radius is specified in inches, the height in feet; volume is required in cubic feet, and the surface area in acres. Several errors are present in the program, all of which would be detected by the units checker.

In the following, comments are made explaining the program, the errors it contains, and how they would be detected. The comments are keyed by line number to the program.

Line Number	Comment
2	All variables in the program that are quantities will be expressed in terms of these basic units.
3	These are the relationships between the units known to the units checker.
5 - 10	Variable radius is in units of inches, height is in units of feet, and so forth.

12 Input values are read into variables radius and height.

13 Lateral surface must be expressed in square feet. (RADIUS/12) is in feet and can be so verified by the checker.

15 Lateral-surface and top-surface are both expressed in square feet, thus their sum is the square feet also. Area is expressed in acres, however, and the checker will issue a message to the effect that though the two sides are commensurate the conversion factor of 43,560 was omitted from the right side of the assignment.

16 The checker will detect that the two sides of the assignment are not commensurate. The right side is in units of $feet^4$, the left is in $feet^3$.

```
(1)     program cylinder (input, output);
(2)     elementary units inches, feet, acre;
(3)     units relationships feet = 12 inches; acre = 43,560 feet^2;
(4)     constant pi = 3.14i5927
(5)     var   radius (*inches*),
(6)           height (*feet*),
(7)           volume (*feet^3*),
(8)           area (*acre*),
(9)           lateral-surface (*feet^2*),
(10)          top-surface (*feet^2*): real;
(11)    begin
(12)          read (radius, height);
(13)          lateral_surface := 2*PI*(RADIUS/12)*height;
(14)          top_surface := PI* ( RADIUS/12)^2
(15)          area := lateral_surface + 2* top_surface;
(16)          volume := PI *(radius^3 *height);
(17)          write (area, volume);
(18)    end;
```

e. Effectiveness. The effectiveness of units checking is limited only by the capabilities of the units processor. Simple unit checkers may only be able to verify that two variables are commensurate, but not determine if proper conversion factors have been applied. That is, a relationship such as 12 inches = feet may not be fully used in checking the computations in a statement, such as line 13 of the example. There we asserted that (radius/12) would be interpreted as converting inches to feet. The checker may not support this kind of analysis however, to avoid ambiguities with expressions such as "one-twelfth of the radius."

f. Applicability. Certain application areas, such as engineering and scientific, often deal with physical units. In others, however, it may be difficult to find analogies to physical units. In particular, if a program deals only in one type of quantity; such as dollars, the technique would not be useful. Units checking can be performed during unit and module testing.

g. Maturity. This technique has been tested and is widely used.

h. User Training. Dimensional analysis is commonly taught in first year college physics on statics; conversion from English to metric units is common throughout society. Direct application of these principles in programming, using a units checker, should require no additional training beyond understanding the capabilities of the specific units checker and the means for specifying units-related information.

i. Cost. If the units checking capabilities is incorporated directly in a compiler, its usage cost should be negligible. If a preprocessor is used, such systems are typically much slower than a compiler (perhaps operating at 1/10 compilation speed), but only a single analysis of the program is required. The analysis is only repeated when the program is changed.

j. References.

(KAR 78) (HEI 82)

4.3.1.2.4 Data Flow Analysis

Data flow analysis determines the presence or absence of those errors that can be

represented as particular sequences of events in a program's execution. This anomaly detection technique is not to be confused with data-flow guided testing (sec. 4.3.1.6), a specialized test technique used primarily for compiler design and optimization.

a. Information Input. Data flow analysis algorithms operate on annotated graph structures that represent the program events and the order in which they can occur. Specifically, two types of graph structures are required: a set of annotated flow graphs and a program invocation (or call) graph. There must be one annotated flow graph for each of the program's procedures. The flowgraph is a digraph whose nodes represent the execution units (usually statements) of the procedures, and whose edges are used to indicate which execution units may follow which other execution units. Each node is annotated with indications of which program events occur as a consequence of its execution. The program invocation (call) graph is also a digraph whose purpose is to indicate which procedures can invoke which others. Its nodes represent the procedures of the program, and its edges represent the invocation relation.

b. Information Output. The output of data flow analysis is a report on the presence of any specified event sequences in the program. If any such sequences are present, then the identity of the sequence is specified with a sample path along which the illegal sequence can occur. The absence of any diagnostic messages concerning the presence of a particular event sequence is a reliable indicator of the absence of any possibility of that sequence.

c. Outline of Method. Data flow analysis relies basically on algorithms from program optimization to determine whether any two particular specified events can occur in sequence. Taking as input a flowgraph annotated with all events of interest, these algorithms focus on two tasks and determine (1) whether there exists some program path along which the two occur in sequence and (2) whether on all program paths the two must occur in sequence. In case one wishes to determine illegal event sequences of length three or more, these basic algorithms can be applied in succession.

A major difficulty arises in the analysis of programs having more than one procedure, because in the case the procedure flowgraphs often cannot be completely annotated prior to data flow analysis. Flowgraph nodes representing procedure invocations must be left either partially or completely unannotated until the flowgraphs of the procedures they

represent have been analyzed. Hence, the order of analysis of the program's procedures is critical. This order is determined by a postorder traversal of the invocation graph, in which the bottom level procedures are visited first, then those which invoke them, and so forth until the main-level procedure is reached. For each procedure, the data flow analysis algorithms must determine the events that can possibly occur both first and last and then make this information available for annotation of all nodes representing invocations of this procedure. Only in this way can it be ensured that any possible illegal event sequence will be determined.

d. Example. The standard example of the application of data flow analysis is the discovery of references to uninitialized program variables. In this case, the program events of interest are the definition of a variable, the reference to a variable, and the undefinition of a variable. Hence, all procedure flowgraphs are annotated to indicate which specific variables are defined, referenced, and undefined at which nodes. Data flow analysis algorithms are then applied to determine whether the undefinition event can be followed by the reference event for any specific variable without any intervening definition event for that variable. If so, a message is produced, indicating the possibility of a reference to an uninitialized variable and a sample program path along which this will occur. A different algorithm is also used to determine whether for a specific variable undefinition must, along all paths, be followed by reference without intervening definition. For invoked procedures, these algorithms are also used to identify which of the procedure's parameters and global variables are sometimes used and always used as inputs and outputs. This information is then used to annotate all nodes representing the invocation of this procedure, to enable analysis of these higher level procedures.

Data flow analysis might also be applied to the detection of illegal sequences of file operations in programs written in languages such as COBOL. Here the operations of interest would be opening, closing, defining (i.e., writing), and referencing (i.e., reading) a file. Errors whose presence or absence could be determined would include: attempting to use an unopened file, attempting to use a closed file, and reading an empty file.

e. Effectiveness. As noted, this technique is capable of determining the absence of event sequence errors from a program, or their presence in a program. When an event sequence error is detected, it is always detected along some specific path. Because these techniques do not study the executability of paths, the error may be detected on an

unexecutable path and hence give rise to a spurious message. Another difficulty is that this technique is not reliable in distinguishing among the individual elements of an array. Hence, arrays are usually treated as if they were simple variables. As a consequence, illegal sequences of operations on specific array elements may be overlooked.

f. Applicability. Data flow analysis is applicable during unit and module testing phases. The applicability of this technique is only limited and restricted by the availability of the (considerable) tools and techniques needed to construct such flowgraphs and call graphs.

g. Maturity. This technique has been proven very effective and efficient and is rapidly appearing as a capability in many state-of-the-art testing tools.

h. User Training. This technique requires only a familiarity with and understanding of the output message. No input data or user interaction is required.

i. Costs. This technique requires computer execution. The algorithms employed, however, are highly efficient, generally executing in time which is linearly proportional to program size. Experience has shown that the construction of the necessary graphs can be a considerable cost factor.

As noted above, no human input or interaction is required, resulting in only the relatively low human cost for interpretation of results.

j. References.

(OST 76) (HEI 82) (GAN 80)
(FOS 76) (MEL 81)

4.3.1.3 Structure Analysis and Documentation

This section contains a description of a test technique that detects errors concerning the control structures (modules, subroutines, branches), and code structures (iterations) of a language. Also, this section contains descriptions of various reports that are generated as a by-product of static analysis.

4.3.1.3.1 Structure Analysis

Application of structure analysis to either code or design allows detection of some types of improper subprogram usage and violation of control flow standards. Structure analysis is also useful in providing required input to data flow analysis tools and is related in principle to two code auditing techniques.

a. Information Input. Two input items are required by structure analysis. The first is the text of the program or design to be analyzed. The text is to be provided to the analyzer in high order language source code, but in some cases in an intermediate form, i.e., after scanning and parsing, but not as object code.

The second input item is a specification of the control flow standards to be checked. These standards are often completely implicit in that they may be part of the rules for programming in the given language or design notation. An example of such a rule is that subprograms may not be called recursively in Fortran. Individual projects may, however, establish additional rules for internal use. Many such rules, for instance limiting the number of lines allowed in a subprogram, can be checked by code auditing. Others, however, can require a slightly more sophisticated analysis and are therefore performed by structure analysis. Two examples in this category are "All control structures must be well nested"; and "Backward jumps out of control structures are disallowed."

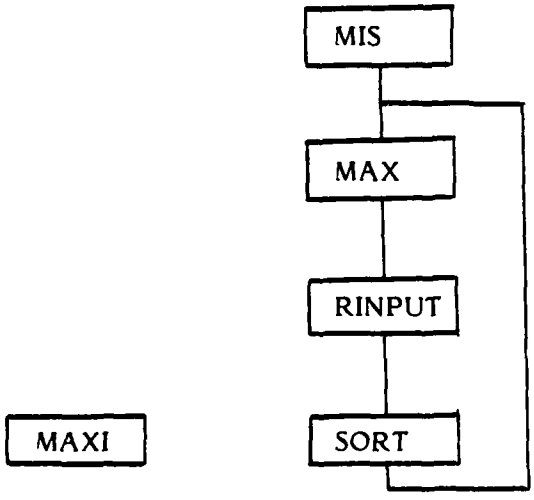
b. Information Output. Error reports and a program call graph are the most common output items of structure analysis. Error reports indicate violations of the standards supplied for the given input. Call graphs indicate the structure of the graph with respect to the use of subprograms: associated with each subprogram is information indicating all routines which call the subprogram and all routines which are called by it. The presence of cycles in the graph (A calls B calls A) indicate possible recursion. Routines which are never called are evident, as well as attempts to call nonexistent routines. Also the presence of structurally dead code is identified.

In checking adherence to control flow standards, this technique will provide a flow graph for each program unit. The flow graph represents the structure of the program, with each control path in the program represented by an edge in the graph.

The flow graph and the call graph are items required as input by data flow analysis, and it is common for the two analysis capabilities to be combined in a single automated tool.

c. Outline of Method. Since structure analysis is an automatable static analysis technique, little user action is required. Aside from providing the input information, the user is only required to peruse the output reports and determine if program changes are required. Some simple manifestations of the tool may not provide detailed analysis reports and therefore rely on the user to examine, for example, the call graph for the presence of cycles.

d. Example. An example of a structural implementation error is as follows: An online management information system program calls a routine MAX to report the largest stock transaction of the day for a given issue. If MAX does not have the necessary information already available, RINPUT is called to read the required data. Since RINPUT reads many transactions for many issues, a sort routine is utilized to aid in organizing the information before returning it to the caller. Due to a keypunch error the sort routine calls routine MAX (instead of MAXI) to aid in the sorting process. This error can be detected as a cycle in the call graph or by locating all disconnected flow-graph components, and will be reported through use of structure analysis.



An example of a violation of control flow standards is as follows. As part of the programming standards formulated for a project, the following rule is adopted: All jumps from within a control structure must be to locations after the end of the structure. The following segment of Pascal contains a violation of this rule which would be reported.

```
100 : X: = 100;
while X  70 do
      begin
        .
        .
        .
      if Z = 5 then goto 100;
        .
        .
        .
      end;
```

e. Effectiveness. Structure analysis is completely reliable for detecting structure errors and violations of the standards specified as input. The standards, however, only cover a small range of programming standards and possible error situations. Thus, this technique is useful only in verifying very coarse program properties. This technique's prime utility therefore is in the early stages of debugging a design or code specification. This technique is useful in detecting unreachable code, infinite loops, and recursive procedure calls, and can also provide helpful documentation on the procedure-calling structure of large programs.

f. Applicability. This technique may be applied from algorithm confirmation through integration test. Particular applicability is indicated in systems involving large numbers of subprograms and/or complex program control flow.

g. Maturity. Structure analysis is automated and has been shown to be effective in detecting structural errors and violations of specified standards.

h. User Training. Minimal training is required for use of the technique. See "Outline of Method."

i. Cost. Little human cost is involved as there is no significant time spent in preparing the input or interpreting the output. Computer resources are small since the processing required can be done very efficiently and only a single run is required for analysis.

j. References.

(FAI 78) (HEI 82) (MEL 81)
(TAY 80B) (GAN 80) (MEL 79)

4.3.1.3.2 Documentation

A by-product of many static analysis techniques are reports produced by automated testing tools. This information can be used to help document the program being analyzed. Descriptions of typical reports are as follows.

a. Global cross-reference report indicating input/output usage for variables in all modules.
b. Module invocations report indicating the calling modules and showing all calling statements.
c. Module interconnection report showing the program's module calling structure.
d. Special global data reports for variables in global areas (e.g., COMMON blocks and COMPOOLs).
e. Program statistics including total size, number of modules, module size distribution, statement type distribution, and complexity measures.
f. Summaries of analysis performed, program statistics, and errors and warnings reported.

References.

(GAN 80) (MEL 81)
(HEI 82) (MEL 79)

4.3.1.4 Program Quality Analysis

This section contains descriptions of complexity ratings and quality measurements. To fully explain the methodology of these topics is beyond the scope of this report. A brief description is supplied here as well as a list of references.

4.3.1.4.1 Halstead's Software Science

Halstead's software science involves several measures of software quality. He combines the disciplines of experimental psychology and computer science into a theory of software complexity. There are four quantities that Halstead's theory utilizes: simple counts of operands and operators, and the total frequencies of operands and operators. With these quantities, Halstead has developed measures for the overall program length, potential smallest volume of an algorithm (the most succinct form in which an algorithm could ever

be expressed), actual volume of an algorithm in a particular language program level (the difficulty of understanding a program), language level (a measure of the relative ease of encoding an algorithm in a specific source language), programming effort (number of elementary discriminations necessary to encode a specific algorithm), program development time, and predicting bug rates in a system.

There is an overall opinion that the Halstead theory on program complexity seems to contain weaknesses, theoretically and practically. Despite these opinions, Halstead metrics have been applied with positive results (FIT 78). However, Hamer and Frewin claim that the "experimental support is largely illusory" (HAM82).

a. Information Input. The input to Halstead's measure is the source code.

b. Information Output. The output of Halstead's measure varies depending on the desired analysis. The above paragraph gives a list of the various measures.

c. Outline of Method. Computing the various measures (metrics) requires counting the number of operands and operators in the algorithm or source code, depending on the metric. It is the vagueness in actually classifying and counting operators and operands that has caused much confusion in interpreting the results of various researchers.

At present, researchers select counting rules that work in their environment. There are no consistent counting rules that work in all environments. Some metrics seem very stable, almost insensitive to the counting rules used to compute them; while others vary widely with very minor counting rule changes.

Halstead's book is notably vague on practical implementation of the theory and Halstead's death occurred before these questions were resolved. For specific information on methodology, it is necessary to examine the work of others who have applied Halstead's metrics as opposed to reading Halstead's book.

d. Example. This section will demonstrate how the level of difficulty metric is applied to a simple program.

The level of difficulty metric asserts that a program with a high level of difficulty is likely to be more difficult to construct and therefore, is likely to be error-prone. The equation for this metric is—

$$L = (2/n1) * (n2/N2),$$

where, L = the level of program difficulty,
n1 = number of unique or distinct operators,
n2 = number of unique or distinct operands,
N2 = total usage of all the operands.

This equation is applied to the following program—

```
C     COMPUTE FIBONACCI SEQUENCE
C     M1 AND M2 ARE CONSECUTIVE TERMS

      M1  =  -1
      M2  =   1
      DO  10   KTR = 1, 30, 2
          M1 = M1 + M2
          M2 = M1 + M2
          PRINT, '(', KTR, ') ', M1
          K = KTR + 1
          PRINT, '(', K, ') ', M2
10        CONTINUE
```

The following table contains the operator and operand counts for the above program—

Operator	j	f(1, j)	Operand	j	f(2, j)
-	1	6	M1	1	5
DO	2	1	M2	2	5
+	3	3	KTR	3	3
PRINT	4	2	K	4	2
CONTINUE	5	1			
	n1=5	N1=13		n2=4	N2=15

Applying this data into the level of difficulty equation results in L = 0.11,

where, f(1,j) = number of occurrences of the jth most frequently occurring operator, where j = 1, 2, ..., n1.

f(2,j) = number of occurrences of the jth most frequently used operand, where j = 1, 2, ..., n2.

In conclusion, the fibonacci program is rated a relatively low level of difficulty (.11); therefore, it follows that the program is simple to construct and is not likely to be error-prone. This is a plausible conclusion given the sample program. For more examples of Halstead's metrics, see HAL77 and HEN81.

e. Effectiveness. A subset of Halstead's measures have shown to be qualitatively effective. A major difficulty with this technique is that there are no standards imposed on the counting rules. Research in this area will yield a more practical and usable methodology.

Halstead's theory has been the subject of considerable evaluative research with such measures as predicting the number of bugs in a program (BEL74, COR76, FIT78, FUM76, OTT79), the time required to implement a program (GOR76, HAL73), debugging time (CUR79, LOV76), and algorithm purity (BUL74, ELS76, HAL73).

f. Applicability. A subset of the various metrics Halstead has proposed have been applied with positive results. It is suggested that future usage of Halstead's metrics should center around those metrics previously studied. This will provide a methodology and guideline to follow with respect to counting rules. This technique is applicable during unit and module test phases.

g. Maturity. Halstead's software science is still in the developmental/experimental stages due to its theoretical weaknesses and the lack of standardized counting rules. However, this technique has been used with varying degrees of results.

h. User Training. To apply Halstead's metrics, knowledge of previous researcher's works is necessary to gain an understanding of the methodology.

i. Costs. This technique is laborious to implement unless it is automated. The counting of operators and operands is a tedious and time consuming task.

j. References.

(BEL 74)	(ELS 76)	(HAL 73)	(OTT 79)
(BUL 74)	(FIT 78)	(HAL 77)	(SHE 83)
(COR 76)	(FUM 76)	(HAM 82)	
(CUR 79)	(GOR 76)	(LOV 76)	

4.3.1.4.2 McCabe's Cyclomatic Number

McCabe's cyclomatic number measures the complexity of a source code module by examining the control flow (IF, WHILE, DO, CASE type statements). It provides a concise means to gauge the number of paths and size of a module of code and therefore locate error-prone sections of code.

a. Information Input. The source code or a detailed design document containing identifiable control flow statements can be the input to this technique.

b. Information Output. The output is the cyclomatic number. This number is an indicator of the complexity of the code. Tools can be used that would provide other insights into the complexity of the code, such as call graphs.

c. Outline of Method. The methodology of this technique involves counting the number of edges, vertices, and connected components of a graph of the program code. The cyclomatic number is then defined as, V(G) = e - n + 2p, where e = no. of edges, n = no. of vertices, and p = no. of connected components. Below is a set of control graphs of constructs found in a structured programming language (MCC76).

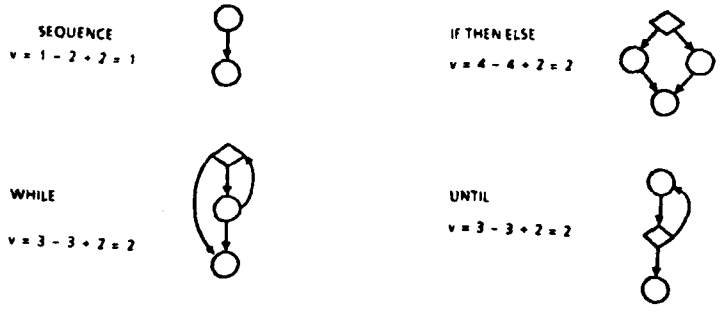
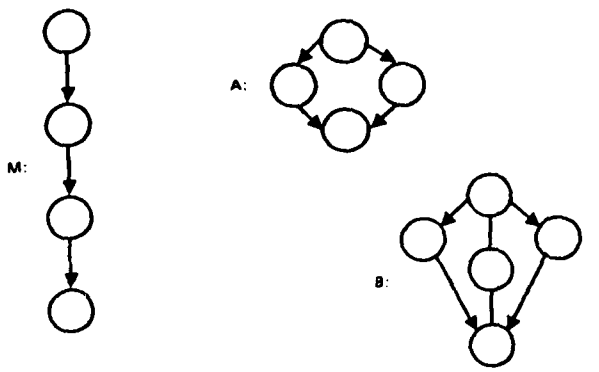


Figure 4.3.1.4-1. Control Graphs of Language Structures

The edges are a count of the lines, the vertices are a count of the circles and decision symbols. The connected components, p, is greater than one when the complexity measure is applied to a collection of programs, for example, if a main program, M, calls subroutines A and B, the control structures would look like:



McCabe has suggested that an upper bound of $V(G) \leq 10$ should be maintained for a module of code. Since the cyclomatic number is based on the examination of control flow, it is therefore a measure of the number of paths through a module of code. By maintaining V(G) at ten or less than ten, control of the number of paths through a module is gained and module testing is facilitated. When $V(G) > 10$, the number of paths through the code should be reduced. The control graph provides a visual picture of the code structure; therefore, it also is helpful in determining where to break up the code.

Two methods to simplify the complexity calculation have been proven. Assuming $p = 1$, $V(G) = Pi + 1$, where Pi = the number of predicates (condition statements). This method requires a count of the predicates in the code. It is not necessary to extract a control graph from the code; however, a control graph is a helpful aid.

The second method involves a visual inspection of the control graph. Given a control graph of a program, a count of the regions is equal to V(G). See MCC76 for details of this method.

All three of the methods to calculate the complexity of a program are useful. In some cases, it is simpler to count the number of predicates, in other cases the regions, or the edges, vertices, and connected components.

d. Example. Given the following example, a count of the number of predicates is the simplest method to calculate the complexity number. This method was applied to a quadratic equation solver program.

No. of Predicates	Program Code
0	C A*X**2 + B*X+C=0
0	C QUADRATIC EQUATION SOLVER
0	READ, A, B, C
1	IF (A .EQ. 0.0) THEN
2	IF (B .EQ. 0.0) THEN
3	IF (C .EQ. 0.0) THEN
0	PRINT, 'TRIVIAL'
0	ELSE
0	PRINT, 'IMPOSSIBLE'
0	END IF
0	ELSE
0	X = - C/B
0	PRINT, 'SINGLE ROOT'
0	PRINT, X
0	END IF
0	ELSE
4	DISC = B**2 -4.0*A*C
0	IF (DISC .GT. 0.0) THEN
0	PRINT, 'REAL ROOTS'
0	X1 = (-B+SQRT(DISC))/(2.0*A)
0	X2 = (-B-SQRT(DISC))/(2.0*A)
0	PRINT, X1, X2
5	END
0	IF (DISC .EQ. 0.0) THEN
0	PRINT, 'MULTIPLE ROOT'
0	X = -B/(2.0*A)
0	PRINT, X
6	END
0	IF (DISC .LT. 0.0) THEN
0	PRINT, 'COMPLEX ROOTS'
0	X = -B/(2.0*A)
0	Y = SQRT(ABS(DISC))/(2.0*A)
0	PRINT, 'X', '+', 'Y', 'I'
0	Y = -Y
0	PRINT, 'X', '+', 'Y', 'I'
0	END
0	END

The cyclomatic number, V(G), is defined as—

$$V(G) = Pi + 1,$$

where, Pi = the number of predicates. This equation applied to the example yields V(G) = 7. Since V(G) for this example is less than ten, the program does not need modification. In other words, the complexity of the program is not great; the number of paths through the code is considered manageable in the context of testing.

A detailed example is provided in MCC76.

e. Effectiveness. The cyclomatic number is effective in determining and controlling the number of program paths in a module of code. There is growing evidence that there exists a direct relationship between code complexity and the number of errors in a segment of code and the time to find and repair such errors. Therefore, it facilitates manageable testing of the code.

f. Applicability. The cyclomatic number can be applied to high-order languages that allow easy detection of control structures (e.g., IF, DO, WHILE, CASE).

When used simultaneously with the coding phase of the software development life cycle, the metric provides an effective way to gauge the size of a module.

g. Maturity. McCabe's complexity measure has progressed beyond the developmental and experimental stage and is widely used. Though there is an indication of a positive relationship with cost and number of errors, this relationship has not been quantitatively studied.

h. User Training. If this technique is automated, there is little work for the user other than interpreting the results. A minimal understanding of McCabe's theory is necessary.

i. Costs. If this technique is not automated, time is spent visually inspecting the code for control structures. Depending on the amount of code to be examined, this could be time consuming.

j. References.
(MCC 76)

4.3.1.4.3 Software Quality Measurement.

Software quality measurement is a method of assessing the quality of software (e.g., reliability, maintainability, testability, correctness, etc.). The metrics are used to measure the presence of the desired qualities at key review points during the development process, and thus allowing periodic prediction of the quality level for the final product.

a. Information Input. The necessary inputs are all tangible software development products (e.g., specifications, standards, code).

b. Information Output. The output of this technique consists of a compliance document, which presents the relationship between the metric scores and the specified requirements. This document is presented at key review points; it provides a picture of the software quality trend over time.

c. Outline of Method. There are two ways to implement software quality metrics. This first method is to apply the metrics during the full scale development of a product. An outline of this methodology is as follows:

- Select quality factors. Software quality metrics can be explained by the following framework.

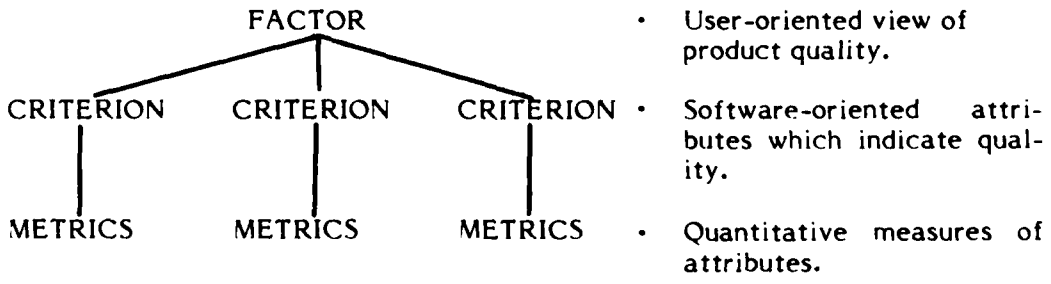


Figure 4.3.1.4-2. Software Quality Metrics Framework

Software quality factors are user-oriented goals (e.g., reliability, maintainability). These factors could be required by contract or a result of some project goal. (Definitions of the software quality factors can be found in (QUA83)).

- Select factor attributes. Depending on the particular system (e.g., uniprocessor, distributed), only a subset of the software factor attributes will apply.

- Establish quantifiable goals. This step requires establishing realistic goals (e.g., a reliability level of 0.98).

The remaining steps of this methodology are described under the second method. This method evaluates a product that has been completed. The following steps are applied regardless of the method chosen.

- Select the software products. Obtain the inputs (i.e., documents and code).

- Select and fill out the metric worksheets. The metric worksheets contain questions concerning specific software criterion (fig. 4.3.1.4-2). For instance, a metric that measures the simplicity criterion asks: Is the design organized in a top-down hierarchical fashion? The metric worksheets are applied to the software products or inputs as previously listed.

- Select scoresheets and score elements, metrics, and criteria factors. Metric worksheets are the inputs to the scoresheets. Scoresheets combine the metrics to form criteria scores; the criteria scores are then combined to form a factor score (fig. 4.3.1.4-2).

- Perform data analysis and generate report. Evaluate the scoring for trends in the data. Unusual scores can be checked for reasonableness at this point.

- Analyze results. The results will indicate whether any corrective actions need to be taken.

d. Example. The methodology can be broken down into a specification phase and an evaluation phase. The first phase involves selecting quality factors and attributes and establishing quantifiable goals. The second phase involves scoring the data within the framework of the specifications and analyzing the results.

Specification Phase

The software quality factor reliability is selected for this example. The definition of reliability is the probability that the software will perform its logical operations in the specified environment without failure. The attributes or criteria associated with the quality factor reliability and the metrics associated with the criteria are listed in the following table.

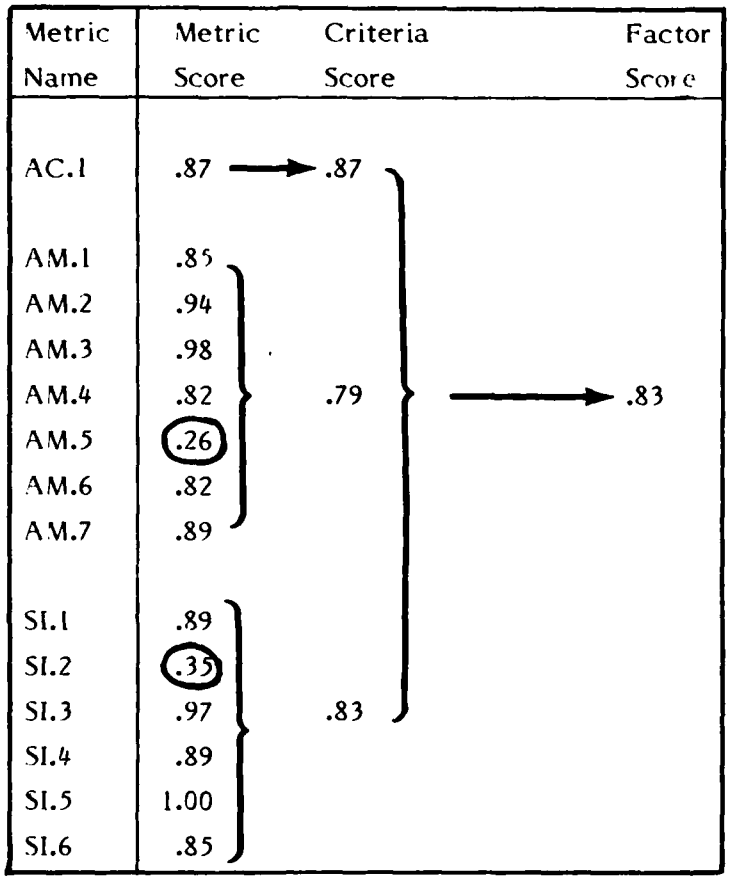
Criteria	Definition
Accuracy (AC)	Those characteristics of the software that provide the required precision in calculations and outputs. The metric associated with accuracy is AC.1 - Accuracy Checklist.
Anomaly Management (AM)	Those characteristics of the software that provide for continuity of operations under and recovery from non-nominal conditions. The metrics associated with anomaly management are-- AM.1 - Error Tolerance/Control Checklist AM.2 - Improper Input Data Checklist AM.3 - Computational Failures Checklist AM.4 - Hardware Faults Checklist AM.5 - Device Errors Checklist AM.6 - Communication Errors Checklist AM.7 - Node/Communication Failures Checklist
Simplicity (SI)	Those characteristics of the software that provide for the definition and implementation of functions in the most non-complex and understandable manner. The metrics associated with simplicity are-- SI.1 - Design Structure Measure SI.2 - Structured Language or Preprocessor SI.3 - Data and Control Flow Complexity Measure SI.4 - Coding Simplicity Measure SI.5 - Specificity Measure SI.6 - Halstead's Level of Difficulty Measure

A quality goal of 0.90 is chosen.

Evaluation Phase

The following steps of the evaluation process involves scoring the metrics, criteria, and the quality factor. This information is given in the following table. Since the measured score for reliability (.83) is less than the desired reliability level of 0.90, corrective

actions need to be taken. With a glance at the data given above, it is obvious that there are two suspect values, AM.5 and SI.2. These unusual scores could result from incorrect metric data or from a poor quality product (software or documentation). In the case where the metric data is correct, then either the product must be improved, or the overall quality goal is unrealistic and must be lowered.



e. Effectiveness. Software quality metrics is an emerging technology. For this reason, this method has had little field use to validate its effectiveness.

f. Applicability. This technique is applicable to a product under development, from algorithm confirmation to verification/CPCI test and to an existing product. Software quality measurement could possibly be applied to the remaining test phases. This is dependent on the particular needs of the user. The metrics are primarily applicable to high-order languages.

g. Maturity. Evaluation of software quality through metrics is very new; therefore, it is not widely used.

h. User Training. An implementor of this technique should be familiar with software design development and coding.

i. Costs. Emphasis on a high software quality goal can sometimes result in cost savings. For example, using metrics to evaluate reliability would result in early detection of errors and, therefore, cost savings. The benefits of some quality factors may not be realized until later in the life cycle (e.g., reusability). An emphasis in reusability is a benefit to future projects.

j. References.

(QUA 83A) (SPE 82) (SOF 83)
(QUA 83B) (SOF 80)

4.3.1.5 Input Space Partitioning

This section contains descriptions of path analysis test techniques. These techniques are characterized by the partitioning of the input space into path domains (subsets of the program input domain) that cause execution of the program's paths. These techniques have been shown to be generators of high quality test data, although current technology limits their use to programs which have a small number of input variables. These techniques are not suitable for widespread use due to the complexity of the methodology; however, less rigorous interpretations of the methodologies are generally used.

4.3.1.5.1 Path Analysis

Path analysis is a technique that generates test data that cause selected paths in a program to be executed. This method presents a procedure for selecting a representative subset of the total set of input data of a program.

a. Information Input. The input needed for this technique is the source code.

b. Information Output. The output of this technique is a set of test data that is sensitive to computation errors, path errors, and missing path errors.

c. Outline of Method. The methodology of path analysis consists of five phases, outlined as follows. See (HOW75) for a detailed discussion of the methodology.

Phase I-Analyze the program and construct descriptions of the standard classes of paths. A given program can have an infinite number of paths through a program. Fortunately, there are strategies that provide a theoretical basis in selecting the best possible subset of paths. To attempt to test the paths is ineffective without a clear idea of which paths comprise a representative set. The boundary-interior method is a strategy for choosing a group of paths through a program to build a finite set of classes in such a way that a test of one path from each class constitutes an intuitively complete set of tests. In this phase, class descriptions of a program are defined using the boundary-interior method. The complete set of class descriptions for a program can be represented in the form of a description tree. Figure 4.3.1.5-2 contains the boundary-interior class description tree for the program in figure 4.3.1.5-1. The leftmost path in the tree describes the class of all paths that test the interior of the loop in the program. The other paths are boundary tests.

```
 1 READ N
 2 IF N < 0 GO TO 10
 3 M ← 1
 4 IF N = 0 GO TO 8
 5 M ← M * N
 6 N ←N - 1
 7 GO TO 4
 8 PRINT M
 9 HALT
10 PRINT -1
11 HALT
```

Figure 4.3.1.5-1. Factorial Program

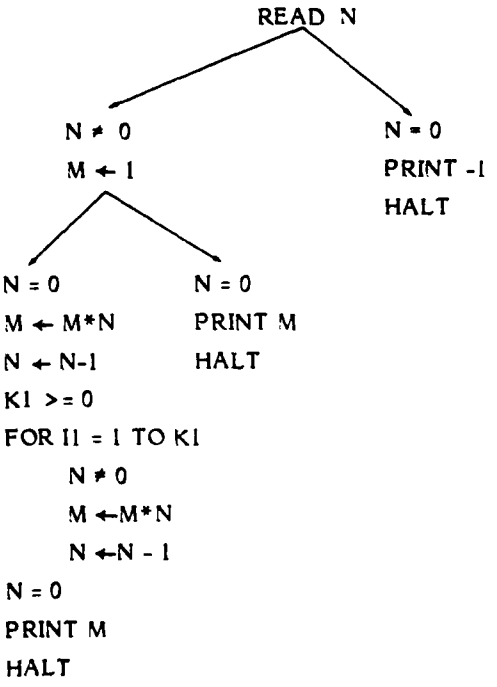


Figure 4.3.1.5-2. Description Tree

Phase II-Construct descriptions of the sets of input data that cause the different standard classes of paths to be followed. The predicates in a program are the condition statements. The predicates in a path, as well as the input and computational statements that affect the variables in the predicates, form an "implicit" description of the subset of the input domain which causes that path to be followed. Figure 4.3.1.5-3 contains the implicit input data description for the interior test class description in Figure 4.3.1.5-2. The input statement "READ N" is substituted by N←# 1, where #1 symbolizes a dummy input variable.

```
N ← #1
N >= 0
N ≠ 0
N ← N - 1
K1 >= 0
FOR I1 = 1 to K1
    N ≠ 0
    N ← N - 1
N = 0
```

Figure 4.3.1.5-3. Implicit Input Data Description

Phase III-Transform the implicit descriptions into equivalent partially explicit descriptions. This phase uses a symbolic interpretation process that attempts to evaluate and delete the assignment statements and FOR-loops in an implicit description. It is not always possible to delete all of the assignment statements from an implicit description; therefore, this phase results in the generation of partial explicit descriptions. The generation of complete explicit descriptions is prevented by the presence of array references and loops in implicit descriptions. Figures 4.3.1.5-4 and 4.3.1.5-5 show the partially explicit description and the explicit descriptions, respectively.

```
#1 >= 0
#1 ≠ 0
N ← #1 - 1
K1 >= 0
FOR I1 = 1 TO K1
    N ≠ 0
    N ← N - 1
N = 0
```

Figure 4.3.1.5-4. Partially Explicit Description

```
#1 >= 0
#1 ≠ 0
K1 >= 0
(#1 - 1 < 0  V  #1 - 1 > K1 - 1)
#1 - 1 -K1 = 0
```

Figure 4.3.1.5-5. Explicit Description

The assignment N←#1 has been evaluated and deleted and the symbolic value #1 is substituted for N in the predicates N = 0 and N ≠ 0. The assignment N←N - 1 has also been evaluated. The symbolic value #1 - 1 of N cannot be substituted for occurrences of N in the For-loop because N is assigned a value in the loop. The assignment N←#1 - 1 must be retained to denote the value of N on entry to the FOR-loop.

The loop in figure. 4.3.1.5-4 can be replaced by the predicate (N < 0 v N > K1 - 1) and the assignment N←N - K1.

Phase IV-Construct explicit descriptions of subsets of input data sets for which the third phase was unable to construct explicit descriptions. Each of the explicit and partially explicit descriptions generated by phase III describes a set of input data. Phase IV constructs explicit descriptions of subsets of the sets that are described by partially explicit descriptions. Subset descriptions are constructed by traversing the FOR-loops in partially explicit descriptions. Partially explicit subset descriptions can be constructed by choosing particular values of K1 in figure 4.3.1.5-4. Figure 4.3.1.5-6 contains the subset description corresponding to the choice of K1 = 0. Figure 4.3.1.5-6 is evaluated to produce explicit subset descriptions (figure 4.3.1.5-7.

```
#1 >= 0
#1 ≠ 0
N ← #1 - 1
N = 0
```

Figure 4.3.1.5-6. Partially Explicit Subset Description

```
#1 >= 0
#1 ≠ 0
#1 - 1 = 0
```

Figure 4.3.1.5-7. Explicit Subset Description

Phase V-Generate the input values that satisfy explicit descriptions. An integrated collection of inequality solution techniques are applied to the complete descriptions to generate test cases. These techniques are applied to both the explicit descriptions, which are generated by phase III of the methodology, and to the subset descriptions generated by phase IV. The explicit subset description in figure 4.3.1.5-7 can be easily solved using a method for linear systems in one variable (figure 4.3.1.5-8).

```
#1  ≥  0
#1  ≠  0        #1 > 0
#1  =  1
thus, #1 = 1
```

Figure 4.3.1.5-8. Explicit Subset Description Solution

d. Example. An example of this technique is given in the preceding section.

e. Effectiveness. This technique detects computation, path, and missing path errors. Only phase I and II have been implemented and has shown to be effective. However, generation of test data cannot be guaranteed for classes of paths containing branch predicates that involve subroutines, functions, or nonlinear expressions in several variables.

f. Applicability. Path analysis is applicable during unit, module, and integration test phase to high-order languages.

g. Maturity. This technique is not highly mature, because it is still in the early stages of its development.

h. User Training. A firm understanding of the underlying methodology is necessary to implement this technique.

i. Costs. Human effort is the main cost factor of this technique.

j. References.
(HOW 75)

4.3.1.5.2 Domain Testing

Domain testing detects errors that occur when a specific input follows the wrong path because of an error in the control flow of the program. The strategy is based on a geometrical analysis of the input domain space and utilizes the fact that points on or near the border of the input space are most sensitive to domain errors.

a. Information Inputs. The program code and minimum-maximum constraints imposed on the input variables are the inputs needed. In most applications, the minimum-maximum range of the input values is known.

b. Information Outputs. This technique detects errors in condition statements (i.e., incorrect relational operator or incorrect operand).

c. Outline of Method. An input space is n-dimensional where n is the number of input variables. The input space structure is a geometrical representation of the input space. Given the inputs and the min-max constraints imposed on the input variables, an input space structure can be constructed. This structure gives a pictorial representation of the domains of the input variables.

Test points are generated for each border segment identified. Border segments are sections of the input space that are delimited by the predicates in the program. These test points determine if both the relational operator and the position of the border are correct.

Those test points that are sensitive to domain testing errors are identified by the following process–

- Choose a predicate in the program that may be in error.
- Determine the assumed correct predicate.
- Compare the outcome of the program with the original predicate with the outcome of the program with assumed correct predicate.
- The test points that result in unequal outcomes are the test points that are sensitive to domain errors.

Fortunately, the number of required test points grows linearly with both the dimensionality of the input space and the number of path predicates. A detailed description of the domain testing strategy methodology can be found in (WHI80).

Given the inputs listed above, an input structure can be drawn. This structure is a graphical perspective of the relationship between the input domain and the control paths through a program. The input-domain is a set of input data points satisfying a path condition.

d. Example. The inclusion of an example of domain testing is beyond the scope of this guidebook due to the complexity of the methodology. See WHI80 for a detailed example.

e. Effectiveness. Domain testing is effective in detecting incorrect relational operators and incorrect operands in condition statements.

f. Applicability. This technique is applicable during the unit or module test phase, and the program should be written in an Algol or Pascal-like language, such that the control structures should be simple and concise.

Arrays, subroutines, and functions are programming language features that are not presently implementable. Further research is needed to determine whether these features pose any fundamental limit to the domain testing strategy.

g. Maturity. Domain testing strategy is still in the developmental or experimental stages and is not widely used for this reason.

h. User Training. There is no specific knowledge needed to do domain testing other than familiarity with the methodology.

i. Costs. The primary costs of the domain testing strategy involves time spent examining and analyzing the test cases. This process could be automated by using some form of an input-output specification.

j. References.
(DEM 83)
(WHI 80)

4.3.1.5.3 Partition Analysis

Partition analysis uses information from both the specification and the implementation to test a procedure. Symbolic evaluation techniques are used to partition the set of input data into subdomains so that the elements of each subdomain are treated uniformly by the specification and the implementation. The information attained from each subdomain is used to guide in generating test data and also verifies consistency between the specification and the implementation.

a. Information Inputs. The necessary inputs are the program specifications written in a formal specification language and the implementation (code).

b. Information Outputs. The outputs constitute a set of test data and a measure of the consistency of the implementation with the specification.

c. Outline of Method. Partition analysis is divided into two types of testing: partition analysis verification, which tests for consistency between the specification and the implementation, and partition analysis testing, which generates test data. An introduction to the terminology is presented to explain the methodology of this testing technique.

The domain of a procedure (or module) can be partitioned into subdomains by examining the specification and the implementation. Symbolic evaluation can be applied to the specification to create subdomains, which are called subspecification (subspec) domains. Each path through the implementation comprises a subdomain called a path domain. As a result, a partition of the whole procedure can be constructed by overlaying these two sets of subdomains. The resulting subdomains are called procedure subdomains.

Partition Analysis Verification. This test provides a means to examine the consistency between the specification and the implementation. There are three properties associated with consistency – compatibility, equivalence, and isomorphism. The specification and the implementation are consistent only if these three properties are shown to be true. Compatibility is shown if the specification and the implementation demonstrate uniformity of declarations of parameters and global variables. The input and output domain of the implementation must agree with that of the specification. Equivalence is shown when the subspec domains are equal to the path domains. Isomorphism is shown if there is a one-to-one correspondence between the subspec domains and the path domains.

Partition Analysis Testing. This technique provides a method of generating test data that are sensitive to computation and domain errors. An examination of the representations of the subspec domains and path domains will expose computation errors. To detect domain errors, an examination of the representations of procedure subdomains is required.

For more detailed information on this methodology, see (RIC 81).

d. Example. An example of partition analysis is beyond the scope of the guidebook due to the complex nature of the methodology. RIC81 provides an excellent and extensive example.

e. Effectiveness. Partition analysis can detect missing path errors, incorrect relational operators in condition statements, domain errors, and computational errors. Initial experimentation with this technique has provided positive results; however, more experimentation is needed to demonstrate the reliability of this method.

f. Applicability. This technique is applicable during unit and module test phases. A formal specification language must be used.

g. Maturity. Partition analysis is not highly mature. It is still in the developmental and research stage. Additional testing is necessary for this technique to become widely used.

h. User Training. Familiarity with symbolic expression of a procedure will prove helpful.

i. Cost. Human effort is the main cost involved in using this technique.

j. References.
(RIC81)

4.3.1.6 Data-Flow Guided Testing

Data-flow guided testing is a method for obtaining structural information about programs that has found wide applicability in compiler design and optimization. Control flow information about a program is then used to construct test sets for the paths to be tested. This specialized technique is not to be confused with data flow analysis (sec. 4.3.1.2), a technique that detects errors that can be represented by particular sequences of events in a program's execution (i.e., reading a file before it is opened).

a. Information Input. A control flow graph representation of the program is the necessary input for this techique.

b. Information Output. Data-flow guided testing will provide information concerning code optimization as follows.

Available Expressions. An expression X opr Y (where, opr is an operator) is "available" at a point p in a flow graph, if every point from the initial node to p evaluates X opr Y, and after the last such evaluation prior to reaching p, there are no subsequent assignments to X or Y. This information enables the elimination of redundant computation of some expressions within each node.

Live Variables. Given a point p in a flow graph, the identification of which variables are "live" at that point, that is, what variables given before this point are used after this point, provides useful information. An important use of this information is evident when object code is generated. After a value is computed in a register, and presumably used within a block, it is not necessary to store that value if it is dead at the end of the block. Also, if all registers are full and another register is needed, using a register with a dead value is ideal since that value does not have to be stored.

Reaching Definitions. Given a definition of a variable, it is desirable to know what uses might be affected by the particular definition. This information can detect potentially undefined names by introducing a dummy definition of each name A preceding the initial node of the flow graph, and seeing whether the dummy definition of A reaches any block that has a use of A and does not define A before that use.

Very Busy Variables. An expression B opr C is very busy at point p if along every path from p we come to a computation of B opr C before any definition of B or C. If B opr C is very busy at p, we can compute it at p, even though it may not be needed there, by introducing the statement T:=B opr C. Then replace all computations A:=B opr C reachable from p by A:=T.

c. Outline of Method. There are two main methods to solving data flow problems: the interval approach (ALL76) and the iterative method (HEC73, KIL73, SCH73). The interval approach collects relevant information by partitioning the program flow graph into subgraphs called intervals, replacing each interval by a single node containing information about the data items used, defined, and preserved within that interval. The iterative approach propagates data flow information in a simple iterative manner until all the

required information is collected. Detail information on these two approaches to data-flow guided testing can be found in KEN76.

d. Example. Examples of solutions to data-flow guided testing problems can be found in HEC75 and ALL76.

e. Effectiveness. This technique has proven to be effective in detecting data flow problems. Examples of these types of problems are: available expressions (COC70, ULL73), live variables (KEN71), reaching definitions (CON70), and very busy variables (SCH73).

f. Applicability. Data-flow guided testing is generally used to assist in the effort of optimizing program code. As a test technique, this method may be utilized during algorithm confirmation, design verification, unit test and module test.

g. Maturity. This technique is highly mature. It is primarily used as a code optimization method.

h. User Training. Familiarity with certain concepts and constructs of graph theory must be acquired in order to implement and understand this technique.

i. Costs. The principal cost associated with this technique involves human effort. Fortunately, a part of this technique has been automated (ALL76), thus human effort can be reduced.

j. References.

(AHO 77)	(CON 70)	(KEN 76)	(ULL 73)
(ALL 76)	(HEC 73)	(KIL 73)	
(COC 70)	(HEC 75)	(SCH 73)	

4.3.2 Dynamic Analysis Techniques

4.3.2.1 Instrumentation-Based Testing

Instrumentation-based testing techniques involve inserting statements or routines that

record properties of the executing program, but do not affect the functional behavior of the program. This section includes technique descriptions of path and structural analysis, performance measurement techniques, assertion testing, and interactive test and debug aids.

4.3.2.1.1 Path and Structural Analysis

Path and structural analysis produces certain analytic information consisting of a listing of the segments of the program undergoing analysis, and the number of times each segment is executed when the program is executed. A segment is a portion of the program that may consist of statements, branches, complete execution paths, or paragraphs as in Cobol.

The goal of path and structural analysis is to increase the amount of code tested. Most times it is impossible to test all the paths or combinations of branches in a large program. It is possible however, to test all branches. For effective testing, all branches and as many paths as possible should be tested.

Instrumentation tools are used to determine how much coverage is achieved in a test run. These tools can also provide timing data, execution traces, and other information. However, the tester himself must formulate input data and decide whether the program has run correctly for each test.

a. Information Inputs. The input to this technique is the source program and an initial set of program test data. In addition, sophisticated coverage analyzers may require input parameters or commands that indicate which of several alternative coverage measures are to be used.

b. Information Outputs. At a minimum, the outputs are a listing of the code that is being analyzed and the number of times a segment is executed when the program is executed.

In addition to coverage information, this analysis may also record and print variable range and subroutine call information. The minimum and maximum values assumed by each variable in a program, the minimum and maximum number of times that loops are iterated during the executions of a loop, and a record of each subroutine call may be reported.

c. Outline of Method. Typically, this analysis consists of two parts—a preprocessor, that inserts code that collects the coverage information during execution. This is called instrumenting the code. The second part is a postprocessor, that contains the capability to reduce data resulting from execution of the instrumented code and prepare reports and test results.

The code that collects coverage data can collect other information as well. The nature of this information depends on the tool used and the level at which the code is instrumented. If probes are inserted after every statement in the program, then the entire history of the execution of a program can be recorded. Of course, instrumenting at the statement level will incur significant computer overhead. To determine branch coverage, it is only necessary to insert probes at every decision statement.

Structural analysis involves the following steps—

- Execution of the preprocessor (input equals the source program to be analyzed) to produce instrumented source code. This step results in the insertion of probes in the program appropriate for test coverage analysis requirements. The probes call special data collection routines or update matrices that record the execution profile of the program.

- Execution of the compiler (input equals the instrumented source code) to produce object code, and the linkage loader to produce the executable program complete with data collection routines.

- Execution of the program, includes execution of probes and data collection routines.

- Execution of the postprocessor to analyze the collected data and to print the test results.

Three levels of structural analysis are possible: statement, branch, and path. Statement analysis, which is least rigorous may be fulfilled 100% without executing all possible program branches. Path analysis on the other hand, requires executing all possible paths in the program, which in some cases may not be achievable. Branch analysis is a usually preferred approach.

There is no automated mechanism for error detection in structural testing. The user must recognize errors by looking at the output of the program. An obvious way to provide automatic assistance in error detection is to use executable assertions. Assertions are better than the execution traces produced by instrumentation tools for two reasons. First, assertions express relationships between variables rather than just reporting their values—in this way error conditions can be checked automatically. Second, execution traces tend to produce large amounts of output, which is wasteful of computer resources and annoying to the user.

When a user finds errors in his program during the course of testing and makes code changes to correct them, the new version of the program must be run through the instrumentation tool again.

d. Example. An example of test coverage analysis is given below.

Application. A quicksort program was constructed which contains a branch to a separate part of the program code that carries out an insertion sort. The quicksort part of the code branches to the insertion sort whenever the size of the original list to be sorted or a section of the original list is below some threshold value. Insertion sorts are more effective than quicksorts for small lists and sections of lists because of the smaller constants in their execution time formulae.

Error. The correct threshold value is 11. Due to a typographical error, the branch to the insertion sort is made whenever the length of the original list, or the section of the list currently being processed is less than or equal to one.

Error discovery. Parts of the insertion sort code are not executed unless the list or list section being sorted is of length greater than one. Examination of the output will reveal that parts of the program are never executed, regardless of the program tests which are used. This will alert and draw the attention of the programmer to the presence of the error. It is interesting to note that this error is not discoverable by the examination of test output data alone since the program will still correctly sort lists.

e. Effectiveness. Structural analysis and use of an instrumentation tool can provide the tester with the following information:

- It can reveal untested parts of a program, so that new test efforts can be concentrated there.

- Data on the frequency of execution of parts of a program, and the time required to execute them, can be tabulated. This information can be used to make the program more efficient through optimization techniques.

- The range of values assumed by a variable (high, low, average, first, last) can be recorded and checked for reasonableness.

- A trace of what has occurred at each statement in a section of code can be printed. This can be useful when debugging.

- The data flow patterns of variables can be analyzed from the execution trace file (HUA79). In this way errors and anomalies in the use of subscripted variables can be detected.

- The degree to which the test cases exercise the structure of the program (traditional testing methods typically exercise 30% to 80% of a program).

Structural analysis is effective in detecting computation errors, logic errors, data handling errors, and data output errors. This technique can help in the detection of omitted program logic, and it can assist in the detection of errors which only occur if a certain combination of segments is executed.

Percentage of errors detected in various programs by structural analysis range from 33% to 92% (HOW78A, HOW80C, MAN74, GAN79, THA76). The combination of structural and functional testing has proven to be effective (THA76).

This technique has been demonstrated as effective on both host and target systems. Refer to CAM81 for details on implementation and use in a target environment.

f. Applicability. This analysis introduces a concept called structure-driven testing. Traditionally, testing has been requirements driven. That is, test cases are developed largely to demonstrate that a program satisfies the functional requirements imposed on it.

Functional or requirement-driven testing should not be superseded by the use of this technique but should be used in conjunction with this technique.

Structural analysis can be used with programs of any type of application. The timing information provided by instrumentation tools is useful in improving the efficiency of time-critical routines.

Structural analysis of large and complex programs is difficult, but these are the programs that most need thorough test coverage. This testing technique has been shown to be effective early in the development life cycle, specifically during unit and module testing phases.

g. Maturity. Path and structural analysis is a highly mature approach to testing a program. Many instrumentation tools have been developed in the last ten years. Tools are available for most programming languages and computers.

h. User Training. There are no special training requirements for the usage of this technique. The use of pre- and post-processors are similar to the use of a compiler.

i. Costs. The most expensive part of structural analysis is the human resources needed to develop test data to achieve a required coverage level and examine output for errors. More experience with structural analysis is needed before good estimates of analysis time and cost can be developed. Available data suggest that using structural analysis to debug programs requires 0.5 to 2.0 person-days per error found.

There is a good deal of data available on the computer overhead of instrumentation tools. The amount of overhead depends on several factors, including the level of instrumentation and the options selected. Generally, instrumentation tools require—

- A 20 - 100% increase in program size.
- A 2 - 50% increase in execution time.

There are numerous instrumentation tools available from Government and commercial sources. Most of the commercial tools sell for less than $10,000.

It is hard to estimate the total costs of structural analysis, because no one knows how to estimate the analysis time or number of test runs required. A user of structural analysis on a large software project claimed a significant cost saving over traditional testing methods.

j. References.

(CAM 81)	(HUA 75)	(GAN 80)	(HOW80C)
(GLA 76)	(HUA 79)	(MEL 79)	(MAN 74)
(HEI 82)	(STU 73)	(MEL 81)	(GAN 79)
(HOW 80A)	(THA 76)	(HOW 78A)	

4.3.2.1.2 Performance Measurement.

There are two types of software performance measurement techniques: execution time and resource analysis and algorithm complexity analysis. These two techniques differ in ease of use and maturity. The former technique is a more practical approach to the measurement of resource usage, whereas the latter technique is a more rigorous approach.

4.3.2.1.2.1 Execution Time and Resource Analysis.

Execution time and resource analysis involves monitoring the execution of a program in order to locate and identify possible areas of inefficiency in the program. Execution data is obtained via a random sampling technique used while the program executes in its normal environment or by the insertion of probes into the program. The probes consist of calls to a monitor that records execution information such as CPU and I/O time. At the end of execution, reports are generated that summarize the resource usage of the program.

a. Information Input. This technique requires as input the program source code and any data necessary for the program to execute.

b. Information Output. The output produced by this technique are reports that show either by statement, groups of statements, and/or module the execution time distribution characteristics and resource usages. For example, information showing per module the number of entries to the module, cumulative execution time, mean execution time per

entry and the percent execution time of the module with respect to the total program execution time.

Other types of reports are as follows—

- A summary of all the sample counts made during data extraction, e.g., the number of samples taken where the program was executing instructions, waiting for the completion of an I/O event, or otherwise blocked from execution.

- A summary of the activity of each load module.

- An instruction location graph that gives the percentage of time spent for each group of instructions partitioned in memory.

- A program timeline that traces the path of control through time.

- A control passing summary that gives the number of times control is passed from one module to another.

- A wait profile showing the number of waits encountered for each group of instructions.

- A paging activity profile that displays pages-in and pages-out for each group of instructions.

- A performance profile showing the amount of CPU, execution time, primary and secondary storage utilized.

c. Outline of Method. Execution time and resource analysis typically consist of two processing units. The first unit runs the program being monitored and collects data concerning the execution characteristics of the program. The second unit reads the collected data and generates reports from it. Two variations in technique implementation are described below.

METHOD 1: This method involves monitoring a program by determining its status at periodic intervals. The period between samples is usually controlled through an elapsed

interval timing facility of the operating system. Samples are taken from the entire address range addressable by the executing task. Each sample may contain an indication of the status of the program, the load module in which the activity was detected, and the absolute location of the instruction being executed. Small sample intervals increase sampling accuracy but result in a corresponding increase in the overhead required by the CPU.

Memory utilization of a user's program can be defined as how well the program uses the memory that has been allocated to the program. In general, poor usage of primary memory occurs when the program requests instructions or data that are not currently resident; this is called a "page fault" because a new "page" of information must be requested from secondary storage. Memory utilization information is obtained by accumulating ordered pairs of page fault counts and program counters. Since the program counter maps into a machine level representation of the program code, only a crude measure of statement locality in the high-level code can be achieved. This information can be displayed using a histogram, where each histogram bar is generated beside a grouping of program statements, showing where poor memory usage is occuring.

The statistics gathered by the data extraction unit is collected and summarized in reports generated by the data analysis unit. References to program locations in these reports will be in terms of absolute addresses. However, in order to relate the absolute locations to source statements in the program, the reports also provide a means to locate in a compiler listing the source statement that corresponds to that instruction. In this way, sources of waits and program locations that use significant amounts of CPU time can be identified directly in the source code; any performance improvements to the program will occur at these identified statements.

METHOD 2: This method involves the insertion of probes (program statements) into the program at various locations of interest. Information, such as CPU time necessary to execute a sequence of statements may be determined in one of two ways. In the first way, the execution of a probe results in a call to a data collection routine which records the CPU clock time at that instant. The execution of a second probe will result in a second call to the data collection routine. A subtraction of the first CPU time from the second will yield the net CPU time utilized. In the second way, the probes are used to record the execution of program statements. Associated with each statement is a

machine dependent estimate of the time required to execute the statement. The execution time estimate is multiplied by the statement execution count to give an estimate of the total time spent executing the statements. This is done for all statements in a program. Reports showing execution time breakdowns by statement, module, statement type, etc. can be produced.

d. Example. Three examples of this technique are given as follows.

Application. A program that solves a set of simultaneous equations is constructed. The program first generates a set of coefficients and a right hand side for the system being solved. It then proceeds to solve the system and output the solution.

Error. In the set of calculations required to solve the system, a row of coefficients is divided by a constant and then subtracted from another row of coefficients. The divisions are performed within a nested DO-loop but should be moved outside the innermost loop, as the dividend and divisors within the loop do not change.

Error discovery. The performance of the program is evaluated through the use of a software monitor. Examination of the output reveals that the program spends almost 85% of its time in a particular address range. Further analysis shows that 16.65% of all CPU time is used by a single instruction. A compiler listing of the program is used to locate the source statement that generated this instruction, which is found to be the statement containing the division instruction. Once the location of the inefficiency is discovered, it is left to the programmer to determine whether and how the code can be optimized.

Application. A particular module in a real time, embedded computer system is required to perform its function within a specific time period. If not, a critical time dependent activity cannot be performed resulting in the loss of the entire system.

Error. The module in question contained an error which involved performing unnecessary comparisons during a table look-up function although the proper table entry was always found.

Error discovery. The problem was discovered during system testing using an execution time analyzer which clearly indicated that the offending module was not able to meet its

performance requirements. The specific error was discovered on further examination of the module.

The following example demonstrates the usage of resource utilization information.

Application. The current work assignment requires that code must be written in Fortran. The application requires the usage of a two-dimensional array for data representation.

Error. A programmer proceeds to program this application as he would in his most familiar programming language C. However, this particular Fortran implementation stores data in matrices by columns, whereas, C implementations store data by rows.

Error Discovery. A histogram of the memory utilization reveals that a huge amount of page faulting occurs in a loop where the two-dimensional array is manipulated. The programmer reviews his code, and modifies his array indices accordingly, so that the left index varies fastest. As a result columns are examined in groups, not rows.

e. Effectiveness. Execution time and resource analysis is a valuable technique in identifying performance problems in a program. The majority of the execution time spent by a program is spent executing a very small percentage of the code. Knowledge of the location of execution time critical code is helpful in optimizing a program in order to satisfy performance requirements and/or reduce costs.

f. Applicability. The value of the technique lies primarily in its use as a performance requirements validation tool. In order to be used to formally validate performance requirements, it is necessary for the performance requirements to have been clearly stated and associated with specific functional requirements. Moreover, the system should be designed so that the functional requirements can be traced to specific system modules. This technique can be applied to any kind of program in any programming language. It is applicable during unit test through verification/CPCI test phase.

g. Maturity. Execution time and resource analysis is widely used and it is highly mature.

h. User Training. There are no special learning requirements for the use of this technique. However, in order to use the technique effectively, the input parameters must

be carefully selected in determining the most relevant reports to be generated. Once the areas of a program which are most inefficient have been identified, it requires skill to modify the program to improve its performance.

i. Costs. The largest cost in using this technique is that incurred by the CPU to extract the data during execution. In one implementation, extraction of data resulted in an increase of user program CPU time by 1 percent to 50 percent (MAR78). Storage requirements also increase in order to provide memory for diagnostic tables and the necessary program modules of the tool.

j. References.

(MAR 78)	(HEI 82)	(GAN 80)
(RAM 75)	(WES 79)	(MEL 81)

4.3.2.1.2.2 Algorithm Complexity Analysis.

Two phases of algorithm complexity analysis can be distinguished: a priori analysis and a posteriori testing. In a priori analysis, a function (of some relevant parameters) is devised that bounds the algorithm's use of time and space to compute an acceptable solution. The analysis assumes a model of computation such as a Turing machine, RAM (random access machine), general-purpose machine, etc. Two general kinds of problems are usually treated (1) analysis of a particular algorithm and (2) analysis of a class of algorithms. In a posteriori testing, actual statistics are collected about the algorithms consumption of time and space, while it is executing.

a. Information Input. The specification of the algorithm and the source code representing the algorithm are necessary inputs.

b. Information Output. The outputs of a priori analysis follow:

- Confidence of algorithms validity.
- Upper and lower computational bounds.
- Prediction of space usage.
- Assessment of optimality.

The output of a posteriori testing is a performance profile.

c. Outline of Method. The methodology of a priori analysis and a posteriori testing is outlined below.

A priori analysis. Algorithms are analyzed with the intention of improving them, if possible and for choosing among several available algorithms. The following criteria may be used:

- Correctness.
- Amount of work done.
- Amount of space used.
- Simplicity.
- Optimality.

Correctness. There are three major steps involved in establishing the correctness of an algorithm. They are as follows.

- Understand that an algorithm is correct if when given a valid input it computes for a finite amount of time and produces the right answer.

- Verify that the mathematical properties of the method and/or formulas used by the algorithm are correct.

- Verify by mathematical argument that the instructions of the algorithm do produce the right answer and do terminate.

Amount of work done. A priori analysis ignores all of the factors which are machine or programming language dependent and concentrates on determining the order of magnitude of the frequency of execution of statements. For denoting the upper bound on an algorithm, the O-notation is used.

Definition, $f(n) = O(g(n))$ if and only if there exists two positive constants C and n_o such that $f(n) \leq C\ g(n)$ for all $n \geq n_o$.

The most common computing times for algorithms—

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) \text{ and } O(2^n).$$

O(1) means that the number of executions of basic operations is fixed and hence the total time is bounded by a constant. The first six orders of magnitude are bounded by a polynomial. However, there is no integer such that n^m bounds 2^n. An algorithm whose computing time has this property is said to require exponential time. There are notations for lower bounds and asymptotic bounds (HOA64). The term, "complexity" is the formal term for the amount of work done, measured by some complexity (or cost) measure.

In general, the amount of work done by an algorithm depends on the size of input. In some cases, the number of operations may depend on the particular input. Some examples of size are:

	Problem	Size of input
1.	Find X in a list of names.	Number of names in the list
2.	Multiply two matrices	Dimensions of the matrices
3.	Solve a system of linear equations	Number of equations and solution vectors

To handle the situation of the input affecting the performance of an algorithm, two approaches (average and worst-case analysis) are used. The average approach assumes a distribution of inputs and then calculates the number of operations performed for each type of input in the distribution and then computes a weighted average. The worst-case approach calculates the maximum number of basic operations performed on any input of a fixed size.

Amount of Space Used. The number of memory cells used by a program, like the number of seconds required to execute a program, depends on the particular implementation. However, some conclusions about space usage can be made by examining an algorithm. A program will require storage space for the instructions, the constants, and variables used by the program, and the input data. It may also use some work space for manipulating the data and storing information needed to carry out its computations. The input data itself may be representable in several forms, some which require more space than others. If the input data has one natural form, for example, an array of numbers or a matrix, then we analyze the extra space used, aside from the program and the input. If the amount of extra space is constant with respect to the input size, the algorithm is said to work "in place".

Simplicity. It is often, though not always, the case that the simplest and most straightforward way of solving a problem is not the most efficient. Yet simplicity in an algorithm is a desirable feature. It may make verifying the correctness of the algorithm easier, and it makes writing, debugging and modifying a program for the algorithm easier. The time needed to produce a debugged program should be considered when choosing an algorithm, but if the program is to be used very often, its efficiency will probably be the determining factor in the choice.

Optimality. Two tasks must be carried out to determine how much work is necessary and sufficient to solve a problem. They are as follows—

- Devise what seems to be an efficient algorithm; call it A. Analyze A and find a function g such that for inputs of size n_1, A does at most g(n) basic operations.

- For some function f, prove a theorem that for any algorithm in the class under consideration, there is some input of size n for which the algorithm must perform at least f(n) basic operations. If the functions g and f are equal, then the algorithm A is optimal.

A posteriori testing. Once an algorithm has been analyzed, the next step is usually to confirm the analysis. The confirmation process consists first of devising a program for the algorithm on a particular computer. After the program is operational, the next step is producing a "performance profile", that is determining the precise amounts of time and storage the program will consume. To determine time consumption, the computer clock is used. Several data sets of varying size are executed and a performance profile is developed and compared with the predicted curve.

A second way to use the computer's timing capability is to take two programs for performing the same task whose orders of magnitude are identical and compare them as they process data. The resulting times will show which, if any, program is faster. Changes to one program which do not alter the order of magnitude but which purport to speed up the program can also be tested in this way.

d. Example. QUICKSORT is a recursive sorting algorithm (HOR78). Roughly speaking, it rearranges the keys and splits the file into two subsections, or subfiles, such that all keys in the first section are smaller than all keys in the second section. Then

QUICKSORT sorts the two subfiles recursively (i.e., by the same method), with the result that the entire file is sorted.

Let A be the array of keys and let m and n be the indices of the first and last entries, respectively, in the subfile that QUICKSORT is currently sorting. Initially m = 1 and n = k. The PARTITION algorithm chooses a key K from the subfile and rearranges the entries, finding an integer j such that for $m \leq i < j$, $A(i) \leq K$; $A(j) = K$; and for $j < i \leq n$, $A(i) \geq K$. K is then in its correct position and is ignored in the subsequent sorting.

QUICKSORT can be described by the following recursive algorithm.

```
QUICKSORT (A,m,n)
if m<n  then do      PARTITION (A,m,n,i,j)
                     QUICKSORT (A,m,j)
                     QUICKSORT (A,i,n)
                     end
```

The PARTITION routine may choose as K any key in the file between A(m) and A(n); for simplicity, let K = A(m). An efficient partitioning algorithm uses two pointers, i and j, initialized to m and n+1, respectively, and begins by copying K elsewhere so that the position A(i) is available for some other entry. The location A(i) is filled by decrementing j until $A(j) \leq K$, and then copying A(j) into A(i). Now A(j) is filled by incrementing i until $A(i) \geq K$, and then copying A(i) into A(j). This procedure continues until the values of i and j meet; then K is put in the last place. Observe that PARTITION compares each key except the original in A(m) to K, so it does n-m comparisons. See (HOA61) for further details.

Worst Case Analysis. If when PARTITION is executed, A(m) is the largest key in the current subfile (that is, $A(m) \geq A(i)$ for $m \leq i \leq n$), then PARTITION will move it to the bottom to position A(n) and partition the file into one section with n-m entries (all but the bottom one) and one section with no entries. All that has been accomplished is moving the maximum entry to the bottom. Similarly, if the smallest entry in the file is in position A(m), PARTITION will simply separate it from the rest of the list, leaving n-m items still to be sorted. Thus if the input is arranged so that each time PARTITION is executed, A(m) is the largest (or the smallest) entry in the section being sorted, then let

p = n-m+1, the number of keys in the unsorted section, then the number of comparisons done is:

$$\sum_{p=2}^{k} (p-1) = \frac{k(k-1)}{2}.$$

Average Behavior Analysis. If a sorting algorithm removes at most one inversion from the permutation of the keys after each comparison, then it must do at least $(n^2 - n)/4$ comparisons on the average. QUICKSORT, however, does not have this restriction. The PARTITION algorithm can move keys across a large section of the entire file, eliminating up to n-2 inversions at one time. QUICKSORT deserves its name because of its average behavior.Consider a situation in which QUICKSORT works quite well. Suppose that each time PARTITION is executed, it splits the file into two roughly equal subfiles. To simplify the computation, assume that $n = 2^p - 1$ for some p. The number of comparisons done by QUICKSORT on a file with n entries under these assumptions is described by the recurrence relation:

$$R(p) = 2^p - 2 + 2\,R(p-1)$$
$$R(1) = 0.$$

The first two terms in R(p), $2^p - 2$, are n-1, the number of comparisons done by PARTITION the first time. The second term is the number of comparisons done by QUICKSORT to sort the two subfiles, each of which has (n-1)/2, or $2^{p-1} - 1$, entries. Expand the recurrence relation to get

$$R(p) = 2^p - 2 + 2R(p-1) = 2^p - 2 + 2(2^{p-1} - 2) + 4\,R(p-2)$$
$$= 2^p - 2 + 2^p - 4 + 2^p - 8 + 8\,R(p-3)$$

thus

$$R(p) = \sum_{i=1}^{p-1} (2^p - 2^i) = (p-1)2^p - \sum_{i=1}^{p-1} 2^i$$

$$= (p-1)2^p - (2^p - 2) = \log n\ (n+1) - n + 1.$$

Thus if A(m) were close to the median each time the file is split, the number of comparisons done by QUICKSORT would be of the order (nlogn). If all permutations of the input data are assumed equally likely, then QUICKSORT does approximately 2nlogn comparisons.

Space Usage. At first glance it may seem that QUICKSORT is an in-place sort. It is not. While the algorithm is working on one subfile, the beginning and ending indexes (call them the borders) of all the other subfiles yet to be sorted must be saved on a stack, and the size of the stack depends on the number of sublists into which the file will be split. This, of course, depends on n. In the worst case, PARTITION may split off one entry at a time in such a way that n pairs of borders are stored on the stack. Thus the amount of space used by the stack is proportional to n.

n	1000	2000	3000	4000	5000
MERGESORT	500	1050	1650	2250	2900
QUICKSORT	400	850	1300	1800	2300

(Time is in milliseconds)

Testing. The results of comparing QUICKSORT and MERGESORT were reported in (HOR78) and are summarized above.

e. Effectiveness. Algorithm analysis has become an important part of computer science. The only issue that limits its effectiveness is that a particular analysis depends on a particular model of computation. If the assumptions of the model are inappropriate, then the analysis suffers.

This technique examines algorithms with the intention of improving them and also provides a means to examine the nature of algorithms (i.e., correctness, amount of work done, amount of space used, simplicity and optimality).

f. Applicability. This technique is applicable during the algorithm confirmation phase through unit test, and can be limited by size of application since the analysis can be lengthy. This technique is most applicable to numerical applications because many of these types of algorithms have been analyzed.

g. Maturity. Algorithm analysis requires highly experienced people with specialized knowledge to implement this technique. It is not a highly used technique.

h. User Training. Algorithm analysis requires significant training in mathematics and computer science. Generally, it will be done by a specialist.

i. Costs. The cost to analyze an algorithm is dependent on the complexity of the algorithm and the amount of understanding about algorithms of the same class.

j. References.

(AHO 74)
(BEN 79)
(HOA 61)
(HOA 64)
(HOR 78)
(WEI 77)

4.3.2.1.3 Executable Assertion Testing.

Executable assertion testing is a three step process consisting of generating the assertions, translating the assertions into processable program statements, and processing the assertions. Assertion generation is a method of capturing the intended functional properties of a program in a special notation, called the assertion language, for insertion into the various levels of the program specification, including the program source code. Once these assertions are identified, they are translated into statements which are compilable. Assertion processing is the process of checking the assertions of a program during execution. This technique serves as a bridge between the more formal program correctness proof approaches and the more common "black box" testing approaches.

Executable assertions are special statements inserted into the source code of a program. They allow the programmer to specify conditions that are required for correct operation of the program. If such a condition does not hold during execution of the program, this fact is reported via an error message. The programmer can also specify actions to be taken when an assertion is violated.

Most compilers do not recognize and translate assertions—an assertion preprocessing tool must be used. The tool generates code, in the same language as the rest of the program, which carries out the condition checking and error handling logic for the assertion. Different preprocessing tools recognize different forms of assertions. A programmer can augment a less powerful tool by writing code to do some of the condition checking.

a. Information Input. A specification of the desired functional properties of the program is the input required for assertion generation. For individual modules, this breaks down, at a minimum, to a specification of the conditions which are "assumed" true on a module

entry and a specification of the conditions desired on module exit. If the specification from which the assertions are to be derived include algorithmic detail, the specifications will indicate conditions which are to hold at intermediate points within the module as well. Generally, assertions are specified in the form of comments in the source program. A program containing user specified assertions is then processed by the dynamic assertion processor.

b. Information Output. The assertions which are created from the functional or algorithmic specifications are expressed in a notation called the assertion language. This notation commonly includes higher level expressive constructs than are found, for example, in the programming language. An example of such a construct is a set. Most commonly the assertion language is equivalent in expressive power to the first order predicate calculus. Thus expressions such as "forall i in set S, A[i] A[i+1] " or "there exists x such that f(x) = 0" are possible. The assertions which are generated, expressing the functional properties of the program, can then be used as input to a dynamic assertion processor, a formal verification tool, walkthrough, specification simulators, and inspections, among other testing techniques. The outputs of assertion processing consists of a list of the assertion checks performed and a list of exception conditions with trace information for determining the nature of the violation.

c. Outline of Method. Executable assertions are constructs added to a programming language. They do two things: indicate by an output message that something has gone wrong in a program, and permit the programmer to specify action that should be taken when such an error occurs. The general form of an executable assertion is:

```
ASSERT condition;
FAIL block;
```

The "condition" is an expression that can be evaluated logically (as TRUE or FALSE) during execution of the program. The "fail block" is optional—it contains the error-handling code.

Assertions must be translated into executable code. This is usually done by a preprocessing tool, although some compilers will accept and translate assertions. The kinds of conditions that can be checked by assertions, and the syntax for declaring these

conditions, vary from tool to tool. The types of assertions accepted by a tool are often referred to as its "assertion language".

The general form of a translated assertion is:

```
IF (NOT condition) THEN
    Print error message;
    Execute fail block
END IF
```

The user must decide what assertion checks to make, encode them in the assertion language, and insert them in the code. Assertions should be "programmed" at the same time as the code itself, for several reasons:

- Writing the assertions increases the programmer's understanding of the purpose and design of the program.

- The assertions themselves will have mistakes which have to be debugged.

- Assertions are useful throughout the life of the program, but may be turned off when the code is introduced for operational use so that run-time efficiency is not affected.

- Adding a full set of assertions to a large, already coded program is a tedious job that no one will want to do.

The processing of the assertion violation will, minimally, keep track of the total number of violations for each assertion, print a message indicating that a violation of the assertion has occurred, and print the values of the variables referenced in the assertion. In addition, the number of times the assertion is checked may be kept and printed when a violation occurs. Sufficient information should be reported upon violation of an assertion to assist the programmer of the specific nature of the error. Specifying assertions within comments is a valuable form of documentation and also ensures that the source program is kept free of non-portable, tool specific directives.

It is important to note that introducing assertions must not alter the functional behavior of a program. Execution time, however, will be increased; the amount of which will depend on the number of assertions which are processed.

How reliable assertions are depends upon the person writing them. To write good assertions, a programmer must understand the way his program is supposed to operate, be familiar with the assertion language, and be thorough in his use of assertions. Assertions have to be debugged just like the rest of a program. In order to effectively utilize assertion processing, test data should be generated which will cause the execution of each assertion.

The test data used has a great affect on the reliability of executable-assertion testing. Test data must cause assertions to be violated or errors will go undetected. To be effective, assertion testing should be combined with a systematic method of generating test data, such as structural or functional testing.

d. Example. Since executable assertion testing is so closely entwined with program development, only brief examples are given. An example of assertion generation and translation is followed by assertion processing. For more thorough examples, see the references.

During program development, the requirement arises for sorting the elements of an array, or table. In order to support flexible processing in the rest of the system, the array is declared with a large, fixed length. However, only a portion of the array has elements in it. The number of elements currently in the array, when passed to the sort routine, is contained in the first element of the array. The array is always to be sorted in ascending order. The sorted array is returned to the calling program through the same formal parameter.

The first specification of the sort routine may appear as follows:

```
      SUBROUTINE SORT ( A, DIM )
C
C     A is the array to be sorted
C     DIM is the dimension of A
C
C
C     sort array
C
      RETURN
      END
```

The characteristics of the subroutine may be partially captured by the following assertions.

```
ASSERT INPUT (0 ≤ A(1) ≤ DIM),(DIM ≥ 2)
ASSERT OUTPUT (A(1) = 0 v A(1) = 1 ∧ true) v
    (A(1)>1  ∧  FORALL I IN [2 . . A(1)]   A(I)   A(I+1))
```

The input assertion notes the required characteristics of A(1) and DIM. The output assertion indicates that if there were 0 or 1 elements in the array, the array is sorted by default. If there are at least 2 elements in the array, then the array is in ascending order.

The next cut at the program may have the following appearance. An intermediate assertion is now shown.

```
      SUBROUTINE SORT ( A, DIM )
C
C     A is the array to be sorted
C     DIM is the dimension of A
C
      ASSERT INPUT (O ≤ A(1)    DIM), (DIM > 2)
      IF (A(1) .LE. 1) GOTO 100
      ASSERT (  2 ≤ A(1) ≤ DIM)
C
C     Sort non-trivial array
C
100   ASSERT OUTPUT (A(1) = O v A(1) = 1     true)
           (A(1)> 1  ∧  FORALL I IN [2 . . A(1)]  A(I) ≤ A(I+1))
      RETURN
      END
```

Suppose a straight selection sort algorithm is chosen for the non-trivial case (i.e., find the smallest element and place it in A(2), find the next smallest and place it in A(3), and so forth, where the original contents of A(2) is exchanged with the element that belongs there in the sorted array). An appropriate intermediate assertion is included within the sorting loop.

```
C
C     PERFORM STRAIGHT SELECTION SORT
      DO 50 J = 2, A(1)
C           find smallest element in A(J) . .  A(A(1)+2)
C           let that element be A(K)
C           exchange A(J) and A(K)
C
            ASSERT (2 ≤ J ≤ A(1))
               (FORALL I IN [2 . . A(1)]   A(I) ≤ A(I+1))
50    CONTINUE
```

A significant issue which we have not dealt with yet is asserting, on termination, that the sorted array is a permutation of the original array. In other words, we wish to assert that in the process of sorting no elements were lost. To do this at the highest level, our first cut at the program requires advanced assertion language facilities. The interested reader is referred to references (CHO, STU75).

The following is an example of assertion processing. The program segment in figure 4.3.2.1-1 is taken from a Pascal program which calls on routine 'sort' to sort array 'A', consisting of 'N' integer elements, in ascending order. The assertion following the call to sort asserts that the elements are indeed in ascending order upon return from the sort procedure. The numbers to the left are the line numbers from the original source.

```
                .
                .
12    Var
13          N : integer;
14          A : array [1..MAXN] of integer;
                    .
                    .
26    begin
                    .
                    .

56                  sort (N,A);
57                  (* assert forall i in [1..N-1] : A[i] <= A[i + 1] *);
```

Figure 4.3.2.1-1. Source Program With Untranslated Assertions

The program segment in figure 4.3.2.1-2 is that which results after all of the assertions have been translated into Pascal. Note that a rather large number of statements were used to implement the assertion. This is due to the rather involved checking required to implement an "assert forall . . .". Simpler assertions will require fewer statements. The specification could be reduced through the use of a common assertion violation procedure.

```
12  Var
13      N : integer;
14      A : array [1..MAXN] of integer;
15      AssertVioCount : array [1.. NumofAsserts] of integer;
16      AssertXqtCount : array [1.. NumofAsserts] of integer;
17      assert : boolean;
            .
            .
29      begin
            .
            .
77          sort (N,A);
78          (* assert forall : in [1..N-1] : A[i] = A[i + 1]*);
78              AssertXqtCount[3] := AssertXqtCount[3]+ 1;
80              assert : = true;
81              i : = 1 ;
82              while (i <= N) and (assert) do (* check assertion *)
83                  if A[i] A[i + 1] then
84                      assert := false
85                  else
86                      i := i + 1;
87              if not assert then begin (* assertion violation *)
88                  AssertVioCount [3] := AssertVioCount [3] + 1;
89                  Writeln ('violation of assertion 3 at statement57');
90                  Writeln ('on execution:', AssertXqtCount [3]);
91                  Writeln ('arrayA =' , A)
92              end (* assertion violation *);
```

Figure 4.3.2.1-2. Translated Assertions

During the testing the following values of A were used in successive executions of routine sort.

Execution	Array A									
1	0	3	12	27	53	171	201	251	390	501
2	0	12	3	53	27	201	171	390	251	501
3	501	390	251	201	171	53	27	12	3	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	100	100	100	999	999	999	1000

The resulting execution produced the following assertion violation:

```
violation of assertion 3 at statement 57 on execution:  3
array A = 3  12  27  53  171  201  251  340  501  0
```

This was the only violation that occurred. Subsequent analysis of the sort procedure indicated that the error was due to an "add-by-one" error on a loop limit. Another detailed example can be found in (STU75).

e. Effectiveness. Executable assertion testing particularly when used in conjunction with an allied technique like functional testing or structural testing, can be extremely effective in aiding the testing of a program. Such effectiveness is only possible, however, when the assertions are used to capture the important functional properties of the program. Assertions such as the following are of no use at all.

```
I = 0
I = I + 1
ASSERT I > 0
```

Capturing the important properties can be a difficult process, and is prone to error. Such effort is well rewarded, though, by increased understanding of the problem to be solved. Indeed, assertion generation is effective because the assertions are to be redundant with the program specifications. This redundancy enables the detection of errors. A cost-effective procedure, therefore, is to develop intermediate assertions only for particularly important parts of the computation. Input and output assertions should always be employed whenever possible.

The effectiveness of executable assertion testing will depend upon the quality of the assertions included in the program being analyzed. Moreover, if the translation is being done by hand, that is, without the use of a dynamic assertion processor, the amount of time required to translate coupled with the unreliability associated with the process will reduce its effectiveness. Nevertheless, the technique can be of significant value in revealing the presence of program errors.

Assertions can be used to detect a wide variety of errors. They are most effective against computation errors, but have also shown to be effective in catching logic, data-input, data handling, interface, data definition, and database errors.

Executable assertions can detect any error that can be expressed as a condition in the assertion language. Some important examples of such errors are:

- The result of a computation is outside of a range of reasonable values, or is inconsistent with another result.

- A variable does not behave as intended: it changes value when it should not, or it does not change in the desired way.

- Control flow is incorrect: the branch taken is incompatible with program conditions, or a special case is not handled properly.

- A call to a routine results in an unacceptable condition on return.

- The output of a routine is incorrect.

As well as detecting errors, executable assertions can do the following things:

- Indicate that a program is operating incorrectly.

- Help the programmer to locate errors.

- Indicate that a program is being used improperly.

- Provide fault-tolerance in a program.

- Express specifications and design intentions as in-line documentation of the program.

- Form the basis for a formal verification of a program.

f. Applicability. This technique is generally applicable during unit, module, and integration test phases. If run-time efficiency is not a problem, the assertions may be left in an activated state in the operational program.

g. Maturity. Tools which process executable assertions exist for most common high-order languages (Cobol, Fortran, Jovial, PL/I, Pascal), since these languages do not include assertions as a statement type. Executable assertion testing is a fairly mature test technique.

h. User Training. To write assertions, a programmer needs to understand how the program is designed and coded. He also needs to develop some skill in handling assertion constructs—this comes with a little experience. To write good assertions, a programmer must then do the following:

- He must find out enough about the application area of the program to develop tight bounds on the values of variables and the results of computations.

- He must write assertions which can trap special error conditions such as logic and data flow errors. This can be difficult when using an assertion language of limited power.

- He must be thorough—all conditions that can be checked for assertions must be identified.

i. Costs. The costs of assertion testing depends on the number of assertions placed in the code, the difficulty of writing and debugging them, and the number of test runs made with the asserted program. There is little data or experience that can be used to gauge the magnitude of these costs. Writing and debugging assertions can be expected to add significantly to costs at the beginning of a project, while the overhead of making test runs should not be as great.

The computer resources used in assertion testing depend on how thoroughly assertions are used. The categories of computer overhead are:

- The time required by the preprocessor and compiler to turn the assertion into executable code.

- The extra execution time required by the assertion checks.

- The extra space required by the source and executable versions of the program due to the assertions.

The preprocessor and compiler overhead will be incurred each time either the assertions or the code are changed. The execution time overhead will be incurred once for each test run. Once a production version of the program is achieved, the assertions can be removed by recompiling the program with the assertions disabled. This is performed by not using the preprocessor, that is, by compiling the source program with the assertions in the form of comments. In this way, the execution time overhead is not incurred by the end-user of the program. The best available estimate of the cost in analyst time to develop the assertions is the cost to write an equivalent amount of code.

j. References.

(BEN 78)	(HET 73)	(STU 75)
(CHO)	(HOA 71)	(TAY 80)
(HEI 82)	(MAN 78)	(YEH 77)

4.3.2.1.4 Interactive Test and Debug Aids.

Interactive test and debug aids are tools used to control and/or analyze the dynamics of a program during execution. The capabilities provided by these tools are used to assist in identifying and isolating program errors. These capabilities allow the user to—

- Suspend program execution at any point to examine program status.
- Interactively dump the values of selected variables and memory locations.
- Modify the computation state of an executing program.
- Trace the control flow of an executing program.

Another common name for this technique is symbolic debugger.

a. Information Input. Interactive test and debug aids require as input the source code that is to be executed and the commands that indicate which testing operations are to be performed by the tool during execution. Included in the commands are indications of which program statements are to be affected by the tool's operation. Commands can be inserted in the source code and/or entered interactively by the user during program execution at preselected break points.

b. Information Output. The information output by an interactive test and debug aid is a display of requested information during the execution of a program. This information may

include the contents of selected storage cells at specific execution points or a display of control flow during execution.

c. Outline of Method. The functions performed by an interactive test and debug aid are determined by the commands input to it. Some common commands are described below.

BREAK: Suspend program execution when a particular statement is executed or a particular variable is altered.

DUMP: Display the contents of specific storage cells, e.g., variables, internal registers, other memory locations.

TRACE: Display control flow during program execution through printed traces of—

- statement executions (using statement labels or line numbers),
- subroutine calls, or
- alterations of a specified variable

SET: Set the value of a specified variable.

CONTENTS: Display the contents of certain variables at the execution of a specific statement.

SAVE: Save the present state of execution.

RESTORE: Restore execution to a previously SAVEd state.

CALL: Invoke a subroutine.

EXECUTE: Resume program execution at a BREAK point.

EXIT: Terminate processing.

These commands allow complete user control over the computation state of an executing program. It allows the tester to inspect or change the value of any variables at any point during execution.

The capabilities of special interactive test and debug aids can also be found in many implementations of compilers for such languages as Fortran, Cobol, and PL/I, which contain testing features added to the language.

d. Example. A critical section of code within a routine is to be tested. The code computes the values of three variables, X, Y, and Z, which later serve as inputs to other routines. To ensure that the values assigned to X, Y, and Z have been correctly computed in this section of code, an interactive test and debug aid is used to test the code.

The code is initially inserted with two commands. A BREAK command is inserted immediately before the first statement and immediately after the last statement of the section of code being tested. Preceding the second BREAK command, a CONTENTS command is also inserted to cause the contents of X, Y, and Z to be displayed after their appropriate values have been assigned.

The program containing the code inserted with these commands is executed. At the first BREAK point, execution is halted and a prompt is issued to the user's terminal requesting a command to be entered. A SAVE command is entered at the terminal to save the present state of execution. A SET command is then entered to set the values of two variables, A and B, which are used to compute the values of X, Y, and Z. The EXECUTE command is entered to resume program execution.

At the end of the execution of the section of code under analysis, the preinserted CONTENTS command displays the computed values of X, Y, and Z. The preinserted BREAK command allows time for these values to be examined and gives the user the opportunity to enter new commands. At this time, a RESTORE command is entered to restore the computation state to the one previously saved by the SAVE command. The computation state returns to that which followed the first BREAK command, allowing the code under analysis to be tested with different input values. Different values for A and B are entered and the contents of X, Y, and Z are observed as before. This process is repeated several times using different, carefully selected values for A and B and the corresponding values of X, Y, and Z are closely examined each time. The results of several computations look suspect. Their input and output values are noted and the code is more thoroughly examined. The program is finally terminated by entering the EXIT command at one of the two possible break points.

e. Effectiveness. To be an effective testing tool, an interactive test and debug aid should be used with a disciplined strategy to guide the testing process. The tools can be easily misused if no testing methodology is associated with their use.

Effectiveness can depend on how easy it is to implement this technique. A desired automatable feature is the ability to trace data from a low-level program representation (object-code) to a high-level program representation (source-code).

f. Applicability. Interactive test and debug aids can be applied to any type of source code. Most existing tools, however, are language dependent (i.e., will operate correctly only for specified languages) and also operating system dependent. This technique is most applicable during unit, module and integration test phases.

g. Maturity. Interactive test and debug aids are highly mature. Tools are available for most high order languages.

h. User Training. A minimal amount of learning is required to use these tools. It is comparable to the learning required in using a text editor. However, if the tool is to be used most efficiently, some learning is required in utilizing the tool in an effective testing strategy.

i. Costs. Programs executing under an interactive test and debug aid will require more computing resources (e.g., execution time, memory for diagnostic tables) than if executed under normal operation. The cost is dependent on the implementation of the tool. For example, those based on interpretive execution will involve costs different from those driven by monitor calls.

j. References.

(MYE 79)
(SPE 79)
(TAY 79)

4.3.2.2 Random Testing

Random testing is a black-box testing technique in which a program is tested by randomly sampling inputs. The sampling criteria may or may not consider the knowledge of the

actual distribution of inputs. This technique is useful in making operational estimates of software reliability.

One striking advantage of random testing is that the test data generator utilized for the random test may create sequences of data and resulting events never envisaged by the designers. Such 'could never happen' events have an unpleasant manner of surfacing as soon as the software project is in the operational environment.

a. Information Input. The input to this technique is the randomly generated input values.

b. Information Output. The primary output of this technique is the detection of errors that occur because certain sequences of events occur during execution of a program.

c. Outline of Method. This ad hoc testing technique involves several steps. Some kind of random input generator is necessary. The type of generator used depends on the type of input data. If the number of outputs is large, then the method of checking for anomalies is usually automated. As the outputs from the random inputs are generated, they are compared with known or expected output values. All discrepancies are reported along with the input values that detect the discrepancy. If the number of outputs is small, then the generated outputs and comparison outputs are compared visually. The investigator then attempts to explain the discrepancies.

d. Example. Random testing may be used to detect errors in the software, as described above, or it may be used in a simulation of the actual environment to estimate actual performance. In the latter case, the random data generator would be designed to provide data according to a 'scenario' which describes the envisioned data environment. Such a scenario would include both valid and invalid data to test the software as severely as possible.

Random testing should never be the principal testing approach in a test plan. It can be, however, a very practical approach in specific environments, and provide a valuable supplement to conventional testing techniques.

e. Effectiveness. This technique can be used to detect a range of errors with a low level of assurance but no specific types of errors with a high level of assurance (fig. 2-14).

However, this method does not detect these errors as well as other test techniques. In terms of probability of detecting errors, a randomly selected collection of test cases has little chance of being an optimal subset of all possible inputs.

f. Applicability. This technique is applicable during the unit, module and integration testing phases, and to a software development project that has a dedicated computer.

g. Maturity. This test technique is straightforward in principle and concept. It is a fairly mature method to test a program.

h. User Training. There are no specific skills needed to implement this technique.

i. Costs. Depending on the environment, the cost of this technique will vary from low to high. Executing a program with the randomly generated inputs is the primary overhead.

j. References.
(MYE 79)

4.3.2.3 Functional Testing

Functional testing is the validation of a program's "functional correctness" by execution under controlled input stimuli. The two techniques described in this section are specification-based functional testing and cause-effect graphing. Cause-effect graphing is a method used to assist specification-based functional testing in the area of test case/test data development.

4.3.2.3.1 Specification-Based Functional Testing.

Functional testing involves generating test data based on knowledge of the functions performed by the program under test, and on the nature of the programs inputs. A large number of test cases can be generated this way for most programs. There are no metrics to indicate the thoroughness of testing or to tell when testing can stop; but certainly every function should be tested at least once.

a. Information Input. Data information and function information are the two inputs of this technique.

Data information. This technique requires the availability of detailed requirements and design specifications and, in particular, detailed descriptions of input data, files and data bases. Both the concrete and abstract properties of all data must be described. Concrete properties include type, value ranges and bounds, record structures, and bounds on file data structure and data base dimensions. Abstract properties include subclasses of data that correspond to different functional capabilities in the system and subcomponents of compound data items that correspond to separate subfunctional activities in the system.

Function information. The requirements and design specifications must also describe the different functions implemented in the system.

Requirements functions correspond to the overall functional capabilities of a system or to subfunctions which are visible at the requirements stage and are necessary to implement overall capabilities. Different overall functional capabilities correspond to conceptually distinct classes of operations that can be carried out using the system. Different kinds of subfunctions can also be identified. Process descriptions in structured specifications, for example, describe data transformations which are visible at requirements time and which correspond to requirements subfunctions. Requirements subfunctions also occur implicitly in data base designs. Data base functions are used to reference, update and create data bases and files.

The designer of a system will have to invent both general and detailed functional constructs in order to implement the functions in the requirements specifications. Structured design techniques are particularly useful for identifying and documenting design functions. Designs are represented as an abstract hierarchy of functions. The functions at the top of the hierarchy denote the overall functional capabilities of a program or system and may correspond to requirements functions. Functions at lower levels correspond to the functional capabilities required to implement the higher level functions. General design functions often correspond to modules or parts of programs which are identified as separate functions by comments. Detailed design functions may be invented during the programming stage of system development and may correspond to single lines of code.

b. Information Output. The output to be examined depends on the nature of the tested function. If it is a straight input/output function, then output values are examined. The testing of other classes of functions may involve the examination of the state of a data base or file.

c. Outline of Method. Functional testing is sometimes referred to as "black box" testing, because detailed information about the program's internal structure need not be used to formulate the test data. Instead, test data is chosen in the following ways:

- Data is chosen to explore whether the program correctly performs the functions that it is intended to perform. The functions should be described in the requirements and specifications for the program.

- The inputs to the program are examined. Using knowledge of the quantities they represent and how the program functions ought to operate on them, the set of possible values for each input variable can be partitioned into "subdomains". Test data sets are generated by taking combinations of samples from each subdomain.

- Some measure of the program's output behavior is defined. Test data is sought which drives this measure toward an undesirable value. Techniques from mathematical optimization can be used to do this.

Errors are detected by manually examining the program's output.

Functional testing is supported by two types of automated tools – test harnesses or drivers and stress testing tools. A test harness provides an environment for testing individual software modules or groups of modules. The tool can fill in for missing program components, including a main program. Test harnesses are most useful in an interactive environment—there they can be used to start, terminate, or interrupt execution at an arbitrary point in a program. Most test harness tools have some debugging capabilities.

A stress testing tool such as the adaptive tester (ref. DAV) can automatically generate test data. The tool tries to find input data that will cause undesirable behavior in the test program. To do this, the user must come up with a numerical measure of program behavior—this is called an "objective function". Various techniques can be used to maximize (or minimize) the value of the objective function; most of these assume that the objective function has certain continuity properties.

d. Examples. This section provides examples of two types of functional testing.

Example 1: Testing of requirements functions.

Application. A computerized dating system was built in which a sequential file of potential dates was maintained. Each client for the service would submit a completed questionnaire which was used to find the five most compatible dates. Certain criteria had to be satisfied before any potential data was selected and it is possible that no date could be found for a client or less than five dates found.

Error. An error in the file processing logic causes the program to select the last potential date in the sequential file whenever there is no potential date for a client.

Error discovery. The number of dates which are found for each client is a dimension of the output data and has extremal values 0 and 5. If the "find-a-date" functional capability of the system is tested over data for a client for which no date should exist then the presence of the error will be revealed.

Example 2: Testing of detailed design functions.

Application. The designer of the computerized dating system in Example 1 decided to process the file of potential dates for a client by reading in the records in sets of 50 records each. A simple function was designed to compute the number of record subsets.

Error. The number of subsets function returns the value 2 when there are less than 50 records in the file.

Error discovery. The error will be discovered if the function is tested over the extremal case for which it should generate the minimal output value 1. Note that this error is not revealed (except by chance) when the program is tested at the requirements specification level. It will also not necessarily be revealed unless the code implementing the function is tested independently and not in combination with the rest of the system.

e. Effectiveness. Studies have been carried out which indicate functional testing to be highly effective in detecting a range of errors in each of the major error categories (fig. 2-14). Its use depends on specific descriptions of system input and output data and a

complete list of all functional capabilities. The method is essentially manual and somewhat informal. If a formal language could be designed for describing all input and output data sets then a tool could be used to check the completeness of these descriptions. Automated generation of extremal, non-extremal and special cases might be difficult since no rigorous procedure has been developed for this purpose.

For many errors it is necessary to consider combinations of extremal, non-extremal and special values for "functionally related" input data variables. In order to avoid combinatorial explosions, combinations must be restricted to a small number of variables. Attempts have been made to identify important combinations (see references) but there are no absolute rules, only suggestions and guidelines.

Since functional testing does not have a well-developed methodology or an objective measure of test thoroughness, its success depends heavily upon the skill of the person conducting the tests. Functional testing often operates under a budget constraint, in which case efficiency in finding errors is of utmost importance.

Any error that prevents a program from operating correctly can be found through functional testing. However, functional testing alone is not useful for determining the efficiency of a program or for debugging. Functional testing cannot guarantee the absence of errors or that the code has been thoroughly tested.

f. Applicability. Functional testing can begin at the unit, module, or integration test phase if the units perform well-defined functions, thus it can be applied during bottom-up development as well as top down.

g. Maturity. Since functional testing does not have a well-developed methodology or an objective measure of test thoroughness, its success depends heavily on the skill of the person conducting the tests. Supporting tools, such as test harnesses or test drivers, are readily available or can easily be implemented.

h. User Training. It is necessary to develop some expertise with the identification of extremal and special cases and to avoid the combinatorial explosions that may occur when combinations of extremal and special values for different data items are considered. It is also necessary to become skilled in the identification of specifications functions although

this process is simplified if a systematic approach is followed for the representation of requirements and design.

i. Costs. The cost of functional testing is most sensitive to the number of test runs made. This is true of both analyst and computer costs, since the set-up costs are low. Test harnesses and stress testing tools have very low overheads and can provide a net saving in computer and analysis costs over manual testing.

j. References.

(HOW 80A)	(HOW 80D)	(DAV)
(HOW 80C)	(MYE 79)	(HEI 82)

4.3.2.3.2 Cause-Effect Graphing.

Cause-effect graphing is a test-case design methodology that can be used with functional testing. It is used to select in a systematic manner a set of test cases which have a high probability of detecting errors that exist in a program. This technique explores the inputs and combinations of input conditions of a program in developing test cases. It is totally unconcerned with the internal behavior or structure of a program. In addition, for each test case derived, the technique identifies the expected outputs. The inputs and outputs of the program are determined through analysis of the requirement specifications. These specifications are then translated into a Boolean logic network or graph. The network is used to derive test cases for the software under analysis.

a. Information Input. The information that is required as input to carry out this technique is a natural language specification of the program that is to be tested. The specification should include all expected inputs and combinations of expected inputs to the program, as well as expected outputs.

b. Information Output. The information output by the process of cause-effect graphing consists of the following:

- An identification of incomplete or inconsistent statements in the requirement specifications.

- A set of input conditions on the software (causes).

- A set of output conditions on the software (effects).

- A Boolean graph that links the input conditions to the output conditions.

- A limited-entry decision table that determines which input conditions will result in each identified output condition.

- A set of test cases.

- The expected program results for each derived test case.

The above outputs represent the result of performing the various steps recommended in cause-effect graphing.

c. Outline of Method. A cause-effect graph is a formal language translated from a natural language specification. The graph itself is represented as a combinatorial logic network. The process of creating a cause-effect graph to derive test cases is described briefly below.

- Identify all requirements of the system and divide them into separate identifiable entities.

- Carefully analyze the requirements to identify all the causes and effects in the specification. A cause is a distinct input condition; an effect is an output condition or system transformation (an effect that an input has on the state of the program or system).

- Assign each cause and effect a unique number.

- Analyze the semantic content of the specification and transform it into a Boolean graph linking the causes and effects; this is the cause-effect graph.

    - Represent each cause and effect by a node identified by its unique number.

    - List all the cause nodes vertically on the left side of a sheet of paper; list the effect nodes on the right side.

- Interconnect the cause and effect nodes by analyzing the semantic content of the specification. Each cause and effect can be in one of two states: true or false. Using Boolean logic, set the possible states of the causes and determine under what conditions each effect will be present.

- Annotate the graph with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.

- By methodically tracing state conditions in the graph, convert the graph into a limited-entry decision table as follows. For each effect, trace back through the graph to find all combinations of causes that will set the effect to be true. Each such combination is represented as a column in the decision table. The state of all other effects should also be determined for each such combination. Each column in the table represents a test case.

- Convert the columns in the decision table into test cases.

This technique to create test cases has not yet been totally automated. However, conversion of the graph to the decision table, the most difficult aspect of the technique, is an algorithmic process which could be automated by a computer program.

d. Example. A database management system requires that each file in the database have its name listed in a master index which identifies the location of each file. The index is divided into ten sections. A small system is being developed which will allow the user to interactively enter a command to display any section of the index at his terminal. Cause-effect graphing is used to develop a set of test cases for the system. The specification for this system is explained in the following paragraphs.

To display one of the ten possible index sections, a command must be entered consisting of a letter and a digit. The first character entered must be a D (for display) or an L (for list) and it must be in column 1. The second character entered must be a digit (0-9) in column 2. If this command occurs, the index section identified by the digit is displayed on the terminal. If the first character is incorrect, error message A is printed. If the second character is incorrect, error message B is printed. The error messages are—

A: INVALID COMMAND

B: INVALID INDEX NUMBER

The causes and effects have been identified as follows. Each has been assigned a unique number.

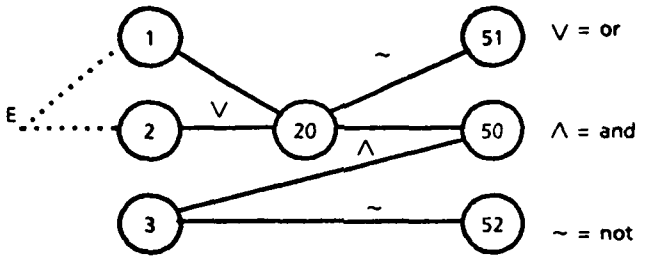
Causes

1. Character in column 1 is D.
2. Character in column 1 is L.
3. Character in column 2 is a digit.

Effects

50. Index section is displayed.
51. Error message A is displayed.
52. Error message B is displayed.

The following Boolean graph is constructed through analysis of the semantic content of the specification.



Node 20 is an intermediate node representing the Boolean state of node 1 or node 2. The state of node 50 is true if the state of nodes 20 and 3 are both true. The state of node 20 is true if the state of node 1 or node 2 is true. The state of node 51 is true if the state of node 20 is not true. The state of node 52 is true if the state of node 3 is not true.

Nodes 1 and 2 are also annotated with a constraint that states that causes 1 and 2 cannot be true simultaneously (the Exclusive constraint). The graph is converted into the following decision table.

	TEST CASES			
CAUSES	1	2	3	4
1	1	0	0	
2	0	1	0	
3	1	1		0
EFFECTS				
50	1	1	0	0
51	0	0	1	0
52	0	0	0	1

Figure 4.3.2.3-1. Decision Table

For each test case, the bottom of the table indicates which effect will be present (indicated by a 1). For each effect, all combinations of causes that will result in the presence of the effect is represented by the entries in the columns of the table. Blanks in the table mean that the state of the cause is irrelevant.

Each column in the decision table is converted into the following test cases.

Test Case No.	Inputs	Expected Results
1	D5	Index section 5 is displayed
2	L4	Index section 4 is displayed
3	B2	INVALID COMMAND
4	DA	INVALID INDEX NUMBER

Figure 4.3.2.3-2. Test Cases

e. Effectiveness. Cause-effect graphing is a technique used to produce a useful set of test cases. It also has the added capability of pointing out incompleteness and ambiguities in the requirement specification.

f. Applicability. Cause-effect graphing can be applied to generate test cases in any type of computing application where the specification is clearly stated and combinations of input conditions can be identified. Manual application of this technique is a somewhat

tedious, long, and moderately complex process. However, the technique could be applied to selected modules where complex conditional logic must be tested. This technique is applicable during algorithm confirmation through unit test phases.

g. Maturity. Cause-effect graphing is a highly mature technique. It is not widely used mainly because it has not been totally automated. Manual application of this technique is time consuming.

h. User Training. Cause-effect graphing is a mathematically-based technique that requires some knowledge of Boolean logic. The requirement specification of the system must also be clearly understood in order to successfully carry out the process.

i. Costs. Manual application of this technique will be highly labor intensive.

j. References.
(ELM 73)
(MYE 76)
(MYE 79)

4.3.2.4 Mutation Testing

Mutation testing is a test technique that involves modifying actual statements of the program. Mutation analysis and error seeding are two examples of this technique.

4.3.2.4.1 Mutation Analysis.

Mutation analysis is a technique for detecting errors in a program and for determining the thoroughness with which the program has been tested. This technique is a method of measuring test data adequacy or the ability of the data to insure that certain errors are not present in the program under test. It entails studying the behavior of a large collection of programs which have been systematically devised from the original program.

a. Information Inputs. The basic input required by mutation analysis is the original source program and a collection of test data sets on which the program operates correctly, and which the user considers to adequately and thoroughly test the program.

b. Information Outputs. The ultimate output of mutation analysis is a collection of test data sets and an assurance that the collection is in fact thoroughly testing the program. It is important to understand that the mutation analysis process may very well have arrived at this final state only after having exposed program errors and inadequacies in the original test data set collection. Hence it is not unreasonable to consider errors detected, new program understanding, and additional test data sets to also be information outputs of the mutation analysis process.

c. Outline of Method. The essential approach taken in the mutation analysis of a program is to produce many versions, each devised by a trivial transformation of the original, and to subject each version to testing by the given collection of test data sets. Because of the nature of the transformations, it is expected that the devised versions will be essentially different programs from the original. Thus the testing regimen should demonstrate that each is in fact different. Failure of execution to produce different results invites suspicion that the collection of test data sets is inadequate. This usually leads to greater understanding of the program and either the detection of errors or an improved collection of test data sets, or both.

A central feature of mutation analysis is the mechanism for creating the program *mutations* - the derived versions of the original program. The set of mutations which is generated and tested is the set of all programs which differ from the original only in a small number (generally 1 or 2) of textual details, such as a change in an operator, variable or constant. Research appears to indicate that larger numbers of changes contribute little or no additional diagnostic power.

The basis for this procedure is the "Competent Programmer" assumptions which state that program errors are not random phenomena, but rather a result of lapses of human memory or concentration. Thus an erroneous program is expected to differ from the correct one only in a small number of details. Hence, if the original program is incorrect, then the set of all programs created by making a small number of textual changes just described, should include the correct program. A thorough collection of test data sets would reveal behavioral differences between the original incorrect program and the derived correct one.

Hence, mutation analysis entails determining whether each mutant program behaves differently from the original program. If so, the mutant is considered incorrect. If not, the mutant must be studied carefully. It is entirely possible that the mutant is in fact functionally equivalent to the original program. If so, its identical behavior is clearly benign. If not, the mutant is highly significant, as it certainly indicates an inadequacy in the collection of test data sets. It may furthermore indicate an error in the original program which previously went undetected because of inadequate testing. Mutation analysis facilitates the detection of such errors by automatically raising the probability of each such error and then demanding justification for concluding that each has not in fact been committed. Most mutations quickly manifest different behavior under exposure to any reasonable test data set collection, and thereby demonstrate the absence of the error corresponding to the mutation by which they were created. This forces detailed attention on those mutants which behave identically to the original and thus forces attention on any actual errors.

If all mutations of the original program reveal different execution behavior, then the program is considered to be adequately tested and correct within the limits of the "Competent Programmer" assumption.

d. Example. Consider the following Fortran program that counts the number of negative and non-negative numbers in array A:

```
      SUBROUTINE COUNT (A, NEG, NONNEG)
      DIMENSION A(5)
      NEG=0
      NONNEG=0
      DO 10 I=1,5

      IF (A(I).GT.0) NONNEG=NONNEG+1
      IF (A(I).LT.0) NEG=NEG+1

10    CONTINUE
      RETURN
      END
```

and the collection of test data sets produced by initializing A in turn to:

I	II	III
1	1	-1
-2	2	-2
3	3	-3
-4	4	-4
5	5	-5

Mutants might be produced based on the following alterations:

Change an occurrence of any variable to any other variable:

e.g.,

A to I

NONNEG to NEG

I to NEG

.

.

.

Change an occurrence of a constant to another constant which is close in value:

e.g.,

1 to 0

0 to 1

0 to -1

1 to 2

Change an occurrence of an operator to another operator:

e.g.,

NEG + 1 to NEG * 1

NEG + 1 to NEG - 1

A(I).GT.0 to A(I).GE.0

A(I).LT.0 to A(I).NE.0

Thus, the set of all "single alteration" mutants would consist of all programs containing exactly one of the above changes. The set of all "double alteration" mutants would consist of all programs containing a pair of the above changes.

Clearly many such mutations are radically different and would quickly manifest obviously different behavior. For example, in changing variable I to A (or vice versa) the program is rendered uncompilable by most compilers. Similarly changing "NEG=0" to "NEG=1" causes a different outcome for test case I.

Significantly, changing A(I).GT.0 to A(I).GE.0 or A(I).LT.0 to A(I).LE.0 produces no difference in run-time behavior on any of the three test data sets. This rivets attention on these mutants, and subsequently on the issue of how to count zero entries. One rapidly realizes that the collection of test data sets was inadequate in that it did not include any zero input values. Had it included one, it would have indicated that:

IF (A(I).GT.0) NONNEG=NONNEG+1
should have been
IF (A(I).GE.0) NONNEG=NONNEG+1.

Thus, mutation analysis has pointed out both this error and this weakness in the collection of test data sets. After changing the program and collection of test data sets all mutants will behave differently which raises our confidence in the correctness of the program.

e. Effectiveness. Mutation analysis is an effective technique for detecting errors, but it must be understood that it requires combining an insightful human with good automated tools. It is a reliable technique for demonstrating the absence of all possible mutation errors (i.e., those involving alteration, interchanging, or omission of operators, variables, etc.)

The need for good tools is easily understood when one realizes that any program has a enormous number of mutations, each of which must be generated, exercised by the test data sets, and evaluated. On the surface, this would appear to entail thousands of edit runs, compilations and executions. Clever tools have been built, however, which operate off a special internal representation of the original program. This representation is readily and efficiently transformed into the various mutations, and also serves as the basis for very rapid simulation of the mutants' executions, thereby avoiding the need for compilation and loading of each mutant.

This tool set still does not bypass the need for humans, however. Humans must still carry out the job of scrutinizing mutants which behave identically to the original program in order to determine whether the mutant is equivalent or whether the collection of test data sets is inadequate.

At the end of a successful mutation analysis, many errors may be uncovered, and the collection of test data sets may be very thorough. Whether the absence of errors is established, however, must be considered relative to the "Competent Programmer" assumption. Under this assumption, clearly all errors of mutation are detectable by mutation analysis, thus the absence of diagnostic messages or findings indicate the absence of these errors. Mutation analysis cannot, however, assure the absence of errors which cannot be modeled as mutations.

f. Applicability. Mutation analysis is applicable to any algorithmic solution specification. This technique is applicable during unit and module test phases. As previously indicated, it can only be considered effective when supported by a body of sophisticated tools. Tools enabling analysis of Fortran and Cobol source text exist. There is furthermore no reason why tools for other coding languages, as well as algorithmic design languages, could not be built.

g. Maturity. Mutation analysis is a fairly mature testing technique. The considerable amount of necessary analyst time to check outputs of mutant programs is its major drawback. Research and development of the technique is continuing, especially to the newer high order languages.

h. User Training. This technique requires the potential mutation analyst to become familiar with the philosophy and goals of this novel approach. In addition it appears that the more familiar the analyst is with the subject algorithmic solution specifications, the more effective he/she will be. This is because the analyst may have to analyze a collection of test data sets to determine how to augment it, and may have to analyze two programs to determine whether they are equivalent.

i. Costs. In view of the previous discussion, it is important to recognize that significant amounts of human analyst time are likely to be necessary to do mutation analysis. The computer time required is not likely to be excessive if the sophisticated tools described earlier are available.

j. References.

(DEM 79) (HEI 82)

(LIP 78)

4.3.2.4.2 Error Seeding.

Error seeding is a technique that evaluates the thoroughness with which a computer program is tested by purposely seeding a supposedly correct program with errors.

a. Information Input. The input required by this technique is the original source program and a collection of test data sets on which the program operates correctly, and which the tester considers to adequately and thoroughly test the program. A desirable second input is information on the relative distribution of different types of errors normally occurring in this type of software.

b. Information Output. The output of this technique is an indication of the level of undetected errors existing in the program. That is, if a large number of the seeded errors are found, it is reasonable to assume that a correspondingly large number of the original, true errors have also been found and corrected.

c. Outline of Method. This technique is similar to mutation analysis (see previous section) in that the basis for the procedure is the "competent programmer" assumptions. These assumptions state that program errors are not a random phenomenon, but rather are the result of lapses of human memory or concentration.

Seeding a program with errors consists of changing statements in the program source code by such alterations as changing an arithmetic operator, changing a relational operator in a condition statement, or substituting the name of one variable for another. It is expected that the introduction of these changes will cause the revised program to generate different test results than the original program. Failure to do so invites suspicion that the test data sets are insensitive. The seeded program and its outputs are usually debugged by persons other than those who seeded the program with errors.

The proportion of seeded errors that remain undetected give an indication of the corresponding proportion of true, undetected errors in the original, unseeded computer

program. That is, if 20% of the seeded errors were not detected by the test data sets, it suggests that 20% of the actual errors have not been detected by this set of tests. The original model is discussed by Harlan Mills (MIL72). Subsequent refinements have been made and are discussed by the various sources listed in the references at the end of this section.

In order to give the error seeding results greater validity, it is desirable to select the seeded errors in a manner consistent with the types of errors expected in the tested computer program.

Several considerations are appropriate (GAN79)—

- To be realistic, the errors should be representative of those found in large programs in both type and frequency of occurrence.

- The error types must be applicable to the software under test and that test environment.

- To evaluate test tools which utilize program execution, one or more errors should lead to abnormal program behavior for at least some test data.

A static analyzer might be used to classify the source statements in the computer program to be tested. The random generation of errors could then be weighted to reflect the corresponding proportion of statement types in the computer program.

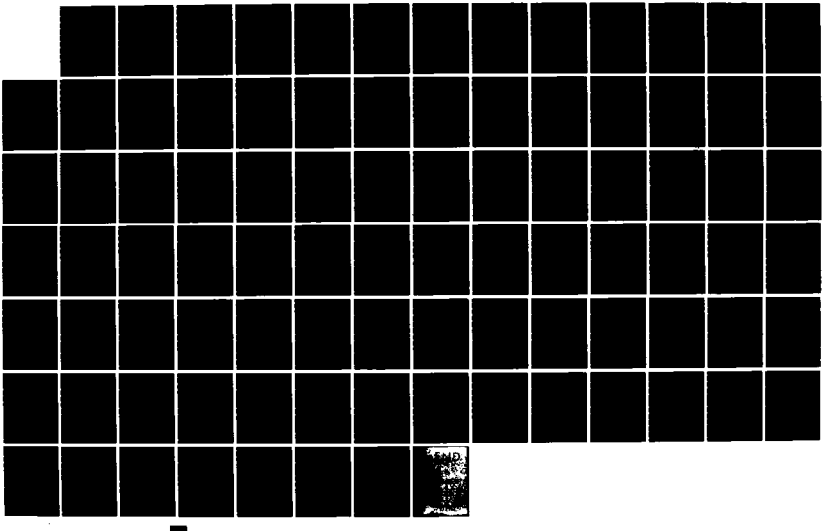
d. Example. The following description shows a typical usage of error seeding in the integration test phase.

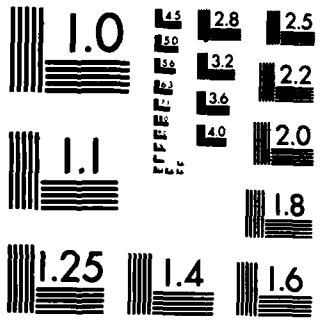
A launch sequence control computer program has completed its integration test successfully. Fifty errors were found and corrected. Historically, such programs have had many residual logical sequencing errors in this group, so it was decided to assess the adequacy of testing for such errors using the error seeding technique.

Using knowledge of the kinds of errors made in the past, a similar assortment of errors were randomly introduced into the launch sequence control. The integration tests were

AD-A147 289 SOFTWARE TEST HANDBOOK: SOFTWARE TEST GUIDEBOOK VOLUME 2(U) BOEING AEROSPACE CO SEATTLE WA ENGINEERING TECHNOLOGY DIV E PRESSON MAR 84 RADC-TR-84-53-VOL-2 3/3

UNCLASSIFIED F30602-82-C-0059 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

then run against the seeded program. Fourteen of the twenty seeded errors were detected; that is, 70% of the seeded errors were found. This indicated that the 50 errors found during integration test were probably only 70% of the total number of original errors. Thus, we would expect an additional 21-22 errors in the code.

The characteristics of the seeded errors that escaped detection were then analyzed and new integration tests were devised to detect those types of errors.

e. Effectiveness. Error seeding can be a reasonably effective technique to assess the adequacy of a set of software tests. The accuracy of the prediction of remaining errors is probably closely related to the realistic distribution of seeded error types and locations. In most cases, the error seeding process should be considered an indicator of remaining errors, not a precise predictor.

f. Applicability. Error seeding is applicable to any type of software in which the presence of residual errors after testing would be intolerable. The more critical the application of the software, the more extensive the application of error seeding. This technique is applicable at the end of unit test, module test, integration test, and, less likely, system test phases. Since error seeding can provide test coverage assessment, it can be applied throughout the testing phases at the discretion of the user.

g. Maturity. Error seeding is a fairly mature theory, though it has not been applied widely as a standard procedure throughout the software development community.

h. User Training. No special knowledge is necessary to implement this technique.

i. Costs. Since the technique requires both test personnel and computer resources to test for the seeded errors after the basic testing is complete, plus follow-on analysis of the test results by the personnel who did the seeding, the cost of the technique would be significant.

It seems unlikely at present that a significant portion of the error seeding technique could be automated, since human judgment is required in most steps of the technique. When expert system techniques become more cost effective, the error seeding approach may be a candidate for automation.

j. References.

(GIL77) (MIL72) (GAN 79)
(HEI82) (MYE79)
(LAP74) (RUD77)

4.3.2.5 Real-Time Testing

Real-time testing may involve testing on "host" computers using environment simulators or testing of the software on the "target" computer in the actual hardware/software system or a simulation of the actual system.

Real-time testing on a host computer may be augmented by the use of a testbed. A testbed is a computer-based test environment used to test a component of software. This test environment simulates the environment under which the software will normally operate. A testbed permits full control of inputs and computer characteristics, allows processing of intermediate outputs without destroying simulated execution time, and allows full test repeatability and diagnostics. To be effective, the controlled circumstances of the testbed must truly represent the behavior of the system of which the software is a part.

In a similar way, real-time testing can be implemented by using the target computer in an artificial "real world". This "real world" is constructed by installing the target computer in a configuration that includes as much of the actual interfacing subsystem hardware as possible. The remainder of the real environment is simulated by one or more computers. This entire configuration is called an "environment simulator," and it has the same goals as a testbed: to provide the most realistic environment in which to test the real-time performance of the software on the embedded computer.

a. Information Input. The information input is the input signals characteristic of the real environment presented in a realistic time sequence. These signals may be generated by the actual interfacing equipment, or by external computers simulating the action of that equipment.

b. Information Output. The information output is the results observed through execution of the software. This information is used as a preliminary means of determining whether

the software will operate as intended in its real environment when that real environment is not available. Timing problems and logic problems, may be exposed.

c. Outline of Method. Both methods unique to the real-time environment provide an environment in which to monitor the operation of software prior to installation in a real system. To be of value, this environment must realistically reflect those properties of the system which will affect or be affected by the operation of the software. However, the environment should simulate only those components in the system which the software requires as a minimum interface with the system. This will permit testing to focus only on the software component for which the testbed is built.

Testbeds are built through the consideration of and proper balance between three major factors:

- The amount of realism required by the testbed to properly reflect the operation of system properties.
- Resources available to build the testbed.
- The ability of the testbed to focus only on the software being tested.

Testbeds come in many forms, depending on the level of testing desired. For single module testing, a testbed may consist merely of test data and a test driver or test harness. A test driver or test harness is a program which feeds input data to the program module being tested, causes the module to be executed, and collects the output generated during the program execution. If a completed, but non-final version of software is to be tested, the testbed may also include stubs. A stub is a dummy routine that simulates the operation of a module that is invoked within a test. Stubs can be as simple as routines that automatically return on a call, or they can be more complicated and return simulated results. The final version of the software may be linked with other software subsystems in a larger total system. The testbed for one component in the system may consist of those system components which directly interface with the component being tested.

As illustrated in the above examples, testbeds permit the testing of a component of a system without requiring the availability of the full, complete system. They merely supply the inputs required by the software component to be executed and provide a repository for outputs to be placed for analysis. In addition, testbeds may contain

monitoring devices which collect and display intermediate outputs during program execution. In this way, testbeds provide the means of observing the operation of software as a component of a system without requiring the availability of other system components, which may be unreliable.

d. Example. An airborne tracking system must be able to process a given rate of radar return messages, apply the required tracking algorithms, and control the displays to radar operators on board the surveillance aircraft. Since actual flight time is prohibitively expensive for all testing, a real time simulator test bed is constructed to simulate the radar and other subsystems with which the airborne operational computer program must perform.

To simulate the environment, a second computer is programmed to provide the radar return messages in real time, and to, in general, simulate the actual environment that the on-board computer would see in the air during a mission. This program in the second computer is the test driver software, and the second computer is called the environment simulator. The second computer also records the outputs of the operational computer program for later analysis. To completely simulate the environment, the environment simulator may also consist of other airborne hardware programmed to present a realistic interface to the operational software being tested.

The test driver then dumps the recorded information from the output file onto a hardcopy device so the output can be analyzed and verified for correctness. It is also possible, when the volume of output data is large, that post-processing software could be written to analyze the test results.

e. Effectiveness. The use of testbeds has proven to be a highly effective and widely used technique to test the operation of software. The use of test drivers or test harnesses, in particular, is one of the most widely used testing techniques. This technique is able to detect a wide range of errors.

f. Applicability. This method is applicable from PQT/FQT through mission test, and for all types of computing applications.

g. Maturity. This technique is widely used and is highly mature.

h. User Training. In order to build an effective testbed, it is necessary to develop a solid understanding of the software and its dynamic operation in a system. This understanding should aid in determining what parts of the testbed deserve the most attention during its construction. In addition, knowledge of the dynamic nature of a program in a system is required in gathering useful intermediate outputs during program execution and in properly examining these results.

i. Cost. The amount of realism desired in a testbed will be the largest factor affecting cost. Building a realistic testbed may require the purchasing of new hardware and the development of additional software in order to properly simulate an entire system. In addition, these added resources may be so specialized that they may seldom, if ever, be used again in other applications. In this way, very sophisticated testbeds may not prove to be highly cost-effective.

j. References.

(HAR 71)
(PAN 78)

4.3.3 Symbolic Testing

Symbolic testing involves the execution of a program from a symbolic point of view. Symbolic testing is applied to programs paths. It can be used to generate expressions which describe the cumulative effect of the computations which occur in a program path. It can also be used to generate a system of predicates which describes the subset of the input domain which causes a specified path to be traversed. The user is expected to verify the correctness of the output which is generated by symbolic execution in the same way that output is verified which has been generated by executing a program over actual values.

a. Information Input. The inputs to this technique are as follows:

Source code. This method requires the availability of the program source code.

Program paths. The path or paths through the program which are to be symbolically evaluated must be specified. The paths may be specified directly by the user or, in some symbolic evaluation systems, selected automatically.

Input values. Symbolic values must be assigned to each of the "input" variables for the path or paths which are to be symbolically evaluated. The user may be responsible for selecting these values or the symbolic evaluation system that is used may select them automatically.

b. Information Output. The outputs of this technique are as follows:

Values of variables. The variables whose final symbolic values are of interest must be specified. Symbolic execution will result in the generation of expressions which describe the values of these variables in terms of the dummy symbolic values assigned to input variables.

System of predicates. Each of the branch-predicates which occur along a program path constrains the input which causes that path to be followed. The symbolically evaluated system of predicates for a path describes the subset of the input domain that causes that path to be followed.

c. Outline of Method. The symbolic execution of a path is carried out by symbolically executing the sequence of assignment statements occurring in the path. Assignment statements are symbolically executed by symbolically evaluating the expressions on the right hand side of the assignment. The resulting symbolic value becomes the new symbolic value of the variable on the left hand side. An arithmetic or logical expression is symbolically executed by substituting the symbolic values of the variables in the expression for the variables.

The branch conditions or branch predicates which occur in conditional branching statements can be symbolically executed to form symbolic predicates. The symbolic system of predicates for a path can be constructed by symbolically executing both assignment statements and branch predicates during the symbolic execution of the path. The symbolic system of predicates consists of the sequences of symbolic predicates that are generated by the execution of the branch predicates.

Symbolic execution systems are used to facilitate symbolic execution. All symbolic execution systems must contain facilities for (1) selecting program paths to be symbolically executed, (2) symbolically executing paths, and (3) generating the required symbolic output.

Three types of path selection techniques have been used: interactive, static and automatic. In the interactive approach, the symbolic execution system is constructed so that control returns to the user each time it is necessary to make a decision as to which branch to take during the symbolic execution of a program. In the static approach the user specifies the paths he wants executed in advance. In the automatic approach the symbolic execution system attempts to execute all those program paths having a consistent symbolic system of predicates. A system of predicates is consistent if it has a solution.

The details of symbolic execution algorithms in different systems are largely technical. Symbolic execution systems may differ in other than technical details in the types of symbolic output they generate. Some systems contain, for example, facilities for solving systems of branch predicates. Such systems are capable of automatically generating test data for selected program paths (i.e., program input data which will cause the path to be followed when the program is executed over that data).

d. Example. An example of symbolic execution follows:

Application. A FORTRAN program called SIN was written to compute the sine function using the McLaurin series.

Errors. The program contained three errors including an uninitialized variable, the use of the expression -1**(I/2) instead of (-1)**(I/2), and the failure to add the last term computed in the series on to the final computed sum.

Different paths through SIN correspond to different numbers of iterations of the loop in the program that is used to compute terms in the series. The symbolic output in the following figure was generated by symbolically evaluating the path that involves exactly three interations of the loop.

PREDICATES:

(X**3/6).GE.E
(X**5/120).GE.E
(X**7/5040).LT.E

OUTPUT:

SIN = ?SUM = (X**3/6) - (X**5/120)

Symbolic output for SIN

Error discovery. The errors in the program are discovered by comparing the symbolic output with the standard formula for the McLaurin series. The symbolic evaluator that was used to generate the output represents the values of variables that have been uninitialized with a question mark and the name of the variable. The error involving the expression (-1)**(I/2) results in the generation of the same rather than alternating signs in the series sum. The failure to use the last computed term can be detected by comparing the predicates for the symbolically evaluated path with the symbolic output value for SIN.

e. Effectiveness. Studies have been carried out which indicate that symbolic evaluation is useful for discovering a variety of errors. This technique is most effective in detecting computation errors but also detects logic errors and data handling errors. This technique is also very effective in assisting in the development of test data.

One of the primary uses of symbolic evaluation is in raising the confidence level of a user in a program. Correct symbolic output expressions confirm to the user that the code carries out the desired computations.

f. Applicability. This method is primarily useful for programs written in languages which involve operations that can be represented in a concise formal way. Most of the symbolic evaluation systems that have been built are for use with algebraic programming languages such as Fortran and PL/1. Algebraic programs involve computations that can be easily represented using arithmetic expressions. It is difficult to generate symbolic output from programs which involve complex operations with "wordy" representations such as the REPLACE and MOVE CORRESPONDING operations in Cobol.

Symbolic execution is most feasible when used with small segments of code. The degree of detail required in its application limits the effectiveness of the technique in testing large scale programs. This technique is applicable during algorithm confirmation, design verification, unit, and module test phases.

g. Maturity. Symbolic evaluation and associated symbolic evaluation systems are in the developmental/experimental phases. Their use requires knowledge of specialized skills. Current tools do not remove enough of the mechanical burden from the user to allow widespread use of the technique.

h. User Training. It takes a certain amount of practice to choose paths and parts of paths for symbolic evaluation. The user must avoid the selection of long paths or parts of paths that result in the generation of expressions that are so large that they are unreadable. If the symbolic evaluation system being used gives the user control over the types of expression simplification that is carried out, then he must learn to use this in a way that results in the generation of the most revealing expressions.

i. Costs. Storage and execution time costs for symbolic evaluation have been calculated in terms of program size, path length, number of program variables and the cost of interpreting (rather than compiling and executing) a program path. The storage required for symbolically evaluating a path of length P in a program with S statements containing N variables is estimated to be on the order of 10(P+S+N) (HOW). Let C1 be the length of a program path, let C2 be the average cost of interpreting a statement in a program path, Exp be the cost of symbolically evaluating a function, Cons be the cost of checking the consistency (i.e., solvability) of a system of symbolic predicates and Cond be the cost of evaluating a condition in a conditional statement. Cons and Cond are expressed in units of the cost of interpreting a statement in a program. The cost (in execution time) of symbolically executing a program path is estimated to be on the order of C1 * C2 (2 + Exp + Cons/10 + Cond/100) (HOW).

j. References.

(HOW 78A) (HEI 82) (CLA 78)
(HOW) (CLA 76)

4.3.4 Formal Analysis

The purpose of formal analysis is to apply the formality and rigor of mathematics to the task of proving the consistency between an algorithmic solution and a rigorous, complete specification of the intent of the solution.

a. Information Input. The two inputs required are the solution specification and the intent specification. The solution specification is algorithmic in form, and is often but not always, executable code. The intent specification is descriptive in form, invariably consisting of assertions, usually expressed in Predicate Calculus.

Additional inputs may be required depending upon the rigor and specific mechanisms to be employed in the consistency proof. For example, the semantics of the language used to express the solution specification are required and must be supplied to a degree of rigor consistent with the rigor of the proof being attempted. Similiarly, simplification rules and rules of inference may be required as input if the proof process is to be completely rigorous.

b. Information Output. The proof process may terminate with a successfully completed proof of consistency, or a demonstration of inconsistency, or it may terminate inconclusively. In the former two cases, the proofs themselves and the proven conclusion are the outputs. In the latter case, any fragmentary chains of successfully proven reasoning are the only meaningful output. Their significance is, as expected, highly variable.

c. Outline of Method. The usual method used in carrying out formal verification is Floyd's *Method of Inductive Assertions* or a variant thereof. This method entails the paritioning of the solution specification into algorithmically straightline fragments by means of strategically placed assertions. This partioning reduces the proof of consistency to the proof of a set of smaller, generally much more manageable lemmas.

Floyd's method dictates that the intent of the solution specification be captured by two assertions - the input assertion, describing the assumptions about the input, and the output assertion describing the transformation of the input which is intended to be the result of the execution of the specified solution. In addition, intermediate assertions must be fashioned and placed within the body of the solution specification in such a way that every loop in the solution specification contains at least one intermediate assertion. Each such intermediate assertion must express completely the transformations which are intended to have occurred or be occurring at the point of placement of the assertion. The purpose of placing the assertions as just described is to assure that every possible program execution is decomposable into a sequence of straightline algorithmic specifications, each of which is bounded on either end by an assertion. If it is known that each terminating assertion is

necessarily implied by executing the specified algorithm under the conditions of the initial assertion, then, by induction it can be shown that entire exeuction behaves as specified by the input/output assertions, and hence as intended. In order for the user to be assured of this, Floyd's Method directs that a set of lemmas be proven. This set consists of one lemma for each pair of assertions which is separated by a straightline algorithmic specification and no intervening other assertion. For such an assertion pair, the lemma states that, under the assumed conditions of the initial assertion, execution of the algorithm specified by the intervening code necessarily implied the conditions of the terminating assertion. Proving all such lemmas establishes what is known as "partial correctness." Partial correctness establishes that whenever the specified solution process terminates, it has behaved as intended. Total correctness is established by, in addition, proving that the specified solution process must always terminate. This is clearly an undecidable question, being equivalent to the Halting Problem, and hence its resolution is invariably approached through the application of heuristics.

In the above procedure, the pivotal capability is clearly the ability to prove the various specified lemmas. This can be done to varying degrees of rigor, resulting in proofs of corresponding varied degrees of reliability and trustworthiness. For the greatest degree of trustworthiness, solution specification, intent specification, and rules of reasoning must all be specified with complete rigor and precision. The principal difficulty here lies in specifying the solution with complete rigor and precision. This entails specifying the semantics of the specification language, and the functioning of any actual execution environment with complete rigor and precision. Such complete details are often difficult or impossible to adduce. They are moreover, when available, generally quite voluminous thereby occasioning the need to prove lemmas which are long and intricate.

d. Example. As an example of what is entailed in a rigorous formal verification activity, consider the specification of a bubble sort procedure. (The details of this can be found in GOO75). The intent of the bubble sort must first be captured by an input/output assertion pair. Next, observing that the bubble solution algorithm contains two nested loops, leads to the conclusion that two additional intermediate assertions might be fashioned, or perhaps one particularly well placed assertion might suffice. In the former case, up to 8 lemmas would then need to be established; one corresponding to each of the (possible two) paths from the initial to each intermediate assertion, one corresponding to each of the two paths from an intermediate assertion back to itself, one for each of the (possibly two)

paths from one intermediate assertion to the other, and finally one for each of the (possibly two) paths from intermediate to terminating assertion. Each lemma would have to be established through rigorous mathematical logic (GOO75). Finally, a proof of necessary termination would need to be fashioned (GOO75).

e. Effectiveness. The effectiveness of formal verification has been attacked on several grounds. First and most fundamentally, formal verification can only establish consistency between intent and solution specification. Hence, inconsistency can indicate error in either or both. The same can be said for most other test techniques, however. What makes this particularly damaging for formal verification is that complete rigor and detail in the intent specification are important, and this requirement for great detail invites error.

The amount of detail also occasions the need for large, complex lemmas. These, especially when proven using complex, detailed rules of inference, produce very large, intricate proofs which are highly prone to error. Complicating this further is the fact that these proofs are generally attempted using the First Order Predicate Calculus. The incompleteness of this mathematical system implies that incorrect theorems cannot be expected to ever be reduced to obvious absurdities. Thus, the prover may, after an arbitrarily long period of unsuccessful effort, not know whether the lemma in question is correct or not.

Finally, formal verification of actual programs is further complicated by the necessity, yet extreme difficulty of rigorously expressing the execution behavior of the actual computing environment which will execute the program. As a consequence of this, the execution environment is generally modelled incompletely and inperfectly, thereby restricting the validity of the proofs in ways which are difficult to determine.

Despite these difficulties, a correctly proven set of lemmas establishing consistency between a complete specification and a solution specification whose semantics are accurately known and expressed, conveys the greatest assurances of correctness obtainable. This ideal of assurance seems best attainable by applying automated theorem provers to design specifications, rather than code.

This technique detects computation and logic errors.

f. Applicability. Formal verification is a technique which can be applied to determine the consistency between any algorithmic solution specification and any intent specification. As elaborated upon earlier, the trustworthiness of the results is highly variable depending primarily upon the rigor with which the specifications are expressed and the proofs are carried out. This technique is applicable during the algorithm confirmation or design verification test phases.

g. Maturity. Formal analysis is still in the research and development phases.

h. User Training. As noted, the essence of this technique is mathematical. Thus, the more mathematical sophistication and expertise which practitioners possess, the better. In particular, a considerable amount of mathematical training and expertise are necessary in order for the results of applying this technique to be significantly reliable or trustworthy.

i. Costs. This technique, when seriously applied must be expected to consume very significant amounts of the time and effort of highly trained mathematically proficient personnel. Hence, considerable human-labor expense must be expected.

As noted earlier, human effectiveness can be considerably improved through the use of automated tools such as thereom provers. It is important to observe, however, that such tools can be prodigious consumers of computer resources. Hence, their operational costs are also quite large.

j. References.

(ELS 72)
(FLO 67)
(GOO 75)

4.4 SUPPORT TECHNIQUES

This section contains descriptions of support techniques. Support techniques facilitate the software testing process; they do not have the capability of detecting specific types of errors when used alone.

4.4.1 Test Data Generators

Test data generators are tools that generate test data to exercise a target program. These tools may generate data through analysis of the program to be tested or through analysis of the expected input in its normal operating environment. Test data generators may use numerical integrators and random number generators to create the data. References: (CLA76) (HOW75) (MIL75) (NAF72).

4.4.2 Test Result Analyzers

Test result analyzers are used in situations where the software cannot be analyzed completely as it is being tested. Typically, output data resulting from execution are written on tape or disk and later analyzed by the software test result analyzer. An example of the need for such postprocessing is the testing of a real-time computer program that must write a history tape of the mission. When run in a simulator, it may be impossible to check the history tape while the simulation test is being run. After the test, a test result analyzer program would read and analyze the history tape to ensure that it was written in the required format and that it correctly recorded the history of the simulated mission. Another example of a test result analyzer is the comparison of the actual results of execution with the expected results. Reference: (PAN78).

4.4.3 Test Management Software

This support software performs a configuration management function by creating a program test library containing the records of all tests, including the test software, test data, software version numbers, and corresponding test results. Such software permits more precise control and recording of the tests and may be appropriate for testing very large computer programs.

4.4.4 Test Completion Criteria Software

This support software uses statistical or reliability prediction techniques in conjunction with user-specified criteria on such factors as cost, desired software product reliability, schedule, and other factors to help determine reasonable test completion criteria. Such software usually incorporates some of the test completion criteria discussed in section

2.7, especially those in approach 4 of that section. Most of this software should be used in an advisory capacity, since the algorithms used are not rigorous.

4.4.5 Test Drivers and Test Harnesses

Test drivers provide the facilities needed to execute a program (e.g., inputs or files, and commands). The input data files for the system must be loaded with data values representing the test situation or events to yield recorded data to evaluate against expected results. Test drivers permit generation of data in external form to be entered automatically into the system at the proper time.

Test harnesses are more generalized test drivers. Test harnesses install the software to be tested in a "test environment," insert stubs to simulate the behavior of missing modules, and provide input data. Some test harnesses are programmable so that they can be tailored by the user to test many kinds of software. Test harnesses usually provide a more complete environment than a test driver, although the distinction between the two is sometimes blurred. Reference: (PAN78).

4.4.6 Comparators

A comparator is a computer program used to compare two versions of source data to determine whether the two versions are identical or to specifically identify where any differences in the versions occur. Comparators are most effective during software testing and maintenance when periodic modifications to the software are anticipated. References: (HET73) (DEC78).

4.5 MISCELLANEOUS TESTING METHODS

The miscellaneous testing methods described in this section have not been included in the test technique taxonomy. Requirements tracing and requirements analysis are beyond the scope of the taxonomy since these techniques facilitate the front-end of the software life cycle. Regression testing is a method of testing that utilizes test techniques from the taxonomy as a means to detect spurious errors resulting from software modifications.

4.5.1 Requirements Tracing.

Requirements tracing provides a means by which to verify that the software of a system addresses each requirement of that system and that the testing of the software produces adequate and appropriate responses to those requirements.

a. Information Input. The information needed to perform requirements tracing consists of a set of system requirements and the software which embodies capability to satisfy the requirements.

b. Information Output. The information output by requirements tracers is the correspondence found between the requirements of a system and the software that is intended to realize these requirements.

c. Outline of Method. Requirements tracing generally serves two major purposes. The first is to ensure that each specified requirement of a system is addressed by an identifiable element of the system software. The second is to ensure that the testing of that software produces results which are adequate responses in satisfying each of these requirements.

A common technique used to assist in making these assurances is the use of test evaluation matrices. These matrices represent a visual scheme of identifying which requirements of a system have been adequately and appropriately addressed and which have not. There are two basic forms of test evaluation matrices. The first form identifies a mapping that exists between the requirement specifications of a system and the modules of that system. This matrix determines whether each requirement is realized by some module in the system, and conversely, whether each module is directly associated with a specific system requirement. If the matrix reveals that a requirement is not addressed by any module, then that requirement has probably been overlooked in the software design activity. If a module does not correspond to any requirement of the system, then that module is superfluous to the system. In either case, the design of the software must be further scrutinized, and the system must be modified accordingly to effect an acceptable requirements-design mapping.

The second form of a test evaluation matrix provides a similar mapping, except the mapping exists between the modules of a system and the set of test cases performed on the system. This matrix determines which modules are invoked by each test case. Used with the previous matrix, it also determines which requirements will be demonstrated to be satisfied by the execution of a particular test case in the test plan. During actual code development, it can be used to determine which requirement specifications will relate to a particular module. In this way, it is possible to have each module print out a message during execution of a test indicating which requirement is referenced by the execution of this module. The code module itself may also contain comments about the applicable requirements.

If these matrices are to be used most effectively in a requirements tracing activity, the two matrices should be used together. The second matrix is built prior to software development. After the software has been developed and the test cases have been designed (based upon this matrix), it is necessary to determine whether the execution of the test plan will actually demonstrate satisfaction of the requirements of the software system. By analyzing the results of each test case, the first matrix can be constructed to determine the relationship that exists between the requirements and software reality.

The first matrix is mainly useful for analyzing the functional requirements of a system. However, the second matrix is also useful in analyzing the performance, interface, and design requirements of the system, in addition to the functional requirements. Both are often used in support of a more general requirements tracing activity, that of preliminary and critical design reviews. This is a procedure used to ensure verification of the traceability of all the above-mentioned requirements to the design of the system. In addition to the use of test evaluation matrices, these design reviews may include the tracing of individual subdivisions in the software design document back to applicable specifications made in the requirements document. This is a constructive technique used to ensure verification of requirements traceability.

d. Example.

- Application. A new payroll system is to be tested. Among the requirements of this system is the specification that all employees of age 65 or older—
  - Receive semi-retirement benefits.
  - Have their social security tax rate readjusted.

To ensure that these particular requirements are appropriately addressed in the system software, test evaluation matrices have been constructed and filled out for the system.

- Error. An omission in the software causes the social security tax rate of individuals of age 65 or older to remain unchanged.

- Error discovery. The test evaluation matrices reveal that the requirement that employees of age 65 or older have their social security tax rate adjusted has not been addressed by the payroll program. No module in the system had been designed to respond to this specification. The software is revised accordingly to accommodate this requirement, and a test evaluation matrix is used to ensure that the added module is tested in the set of test cases for the system.

e. Effectiveness. Requirements tracing is a highly effective technique in discovering errors during the design and coding phases of software development. This technique has proven to be a valuable aid in verifying the completeness, consistency, and testability of software. If a system requirement is modified, it also provides much assistance in retesting software by clearly indicating which modules must be rewritten and retested. Requirements tracing can be a very effective technique in detecting errors early in the software development cycle which could otherwise prove to be very expensive if discovered later.

f. Applicability. This technique is generally applicable in large or small system testing and for all types of computing applications during the design and code phases. However, if the system requirements themselves are not clearly specified and documented, proper requirements tracing can be very difficult to accomplish in any application.

g. Maturity. Requirements tracing is a highly mature technique; it is widely used and has proven to be effective.

h. User Training. Knowledge and a clear understanding of the requirements of the system is essential. More complex systems will result in a corresponding increase in required learning.

i. Costs. No special tools or equipment are needed to carry out this technique. The major cost in requirements tracing is that associated with human labor expended.

j. References.

(HET 76)
(THR 75)

4.5.2 Requirements Analysis.

The requirements for a system will normally be specified using some formal language which may be graphical and/or textual in nature. Requirements analysis checks for syntactical errors in the requirements specifications and then produces a useful analysis of the relationships between system inputs, outputs, processes, and data. Logical inconsistencies or ambiguities in the specifications can also be identified by requirements analysis.

a. Information Input. The form and content of the input will vary greatly for different requirements languages. Generally, there will be requirements regarding what the system must produce (outputs) and what types of inputs it must accept. There will usually be specifications describing the types of processes or functions which the system must apply to the inputs in order to produce the outputs. Additional requirements may concern timing and volume of inputs, outputs, and processes as well as performance measures regarding such things as response time and reliability of operations. The form of the inputs to requirements analysis is specified by the requirements specification language and varies considerably for different languages. In some cases all inputs are textual, whereas some languages utilize all graphical inputs from a display terminal (e.g., boxes might represent processes and arrows between boxes might represent information flow).

b. Information Output. Nearly all analysis produces error reports showing syntactical errors or inconsistencies in the specifications. For example, the syntax may require that the outputs from a process at one level of system decomposition must include all outputs from a decomposition of that process at a more detailed level. Similarly, for each system output there should be a process which produces that output. Any deviations from these rules would result in error diagnostics.

Each requirements analysis produces a representation of the system which indicates static relationships among system inputs, outputs, processes, and data. The analysis also represents dynamic relationships and provides an analysis of them. This may be a precedence relationship, e.g., process A must execute before process B. It may also include information regarding how often a given process must execute in order to produce the volume of output required. This technique produces a detailed representation of relationships between different data items. This output can sometimes be used for developing a data base for the system. Requirements analysis goes even further and provides a mechanism for simulating the requirements using the generated system representation including the performance and timing requirements.

c. Outline of Method. The user must provide the requirements specifications as input for the analysis. The analysis is carried out in an automated manner and provides it to the user who must then interpret the results. Often the user can request selected types of outputs, e.g., an alphabetical list of all the processes or a list of all the data items of a given type. This analysis can be implemented either interactively or in a batch mode. Once the requirements specifications are considered acceptable, requirements analysis provides the capability for simulating the requirements. It is necessary that the data structure and data values generated from the requirements specifications be used as input to the simulation, otherwise the simulation may not truly represent the requirements.

d. Example. Suppose that a process called PROCESS B produces two files named H2 and H3 from an input file named M2. (The purposes of the files are irrelevant to the discussion.) Suppose also that PROCESS D accepts Files H2 and H3 as input and produces Files J3 and J6 output. In addition, PROCESS G is a subprocess of PROCESS D and it accepts File H3 as input and produces File J6. Then the following pseudo specification statements might be used to describe the requirements. (Note that these requirements are close to design, but this is often the case.)

```
PROCESS B
      .
      .
      .
USES FILE M2
PRODUCES FILES H2, H3

PROCESS D
      .
      .
      .
USES FILES H2, H3
PRODUCES FILES J3, J6

PROCESS G
      .
      .
      .
SUBPROCESS OF PROCESS D
USES FILE H3
PRODUCES FILE J6
```

The requirements specification imply a certain precedence of operations, e.g., PROCESS D cannot execute until PROCESS B has produced files H2 and H3. Detailed descriptions of what each process does would normally be included, but are omitted for brevity. Requirements analysis would probably generate a diagnostic since the statement for PROCESS D fails to indicate that it includes the subprocess G. A diagnostic would also be generated unless there are other statements which specify that file M2, needed by PROCESS B, is available as an existing file or else is produced by some other process. Similarly, other processes must be specified which use files J3 and J6 as input unless they are specified as files to be output from the system. Otherwise, additional diagnostics would be generated. It can be seen that some of the checks are similar to data flow analysis for a computer program. However, for large systems the analysis of requirements becomes very complex if requirements for timing and performance are included, and if timing and volume analysis are to be carried out. (Volume analysis is concerned with such things as how often various processes must execute if the system is to accept and/or produce a specified volume of data in a single given period of time.)

e. Effectiveness. Requirements analysis is very effective for maintaining accurate requirements specifications. For large systems with a large number of requirements it is essential. On the other hand, most existing requirements analysis tools are rather expensive to obtain and use, and they may not be cost effective for development of small systems.

f. Applicability. Requirements analysis is applicable for use in developing most systems. It is particularly useful for analysis of requirements for large and complex systems.

g. Maturity. Requirements analysis is generally used on large, complex systems as opposed to small systems.

h. User Training. Requirements analysis generally require a considerable amount of training of personnel.

i. Cost. Most requirements analysis tools are expensive to obtain and use. They generally require a large amount of storage within a computer and so can only be used on large computers.

j. References.
(ALF 76)
(TEI 72)

4.5.3 Regression Testing.

Regression testing is a technique that detects spurious errors caused by software modifications or corrections.

a. Information Input. Regression testing requires that a set of software test cases be maintained and available throughout the entire life of the software. The test cases should be complete enough so that all of the software's functional capabilities are thoroughly tested. If available, acceptance tests should be used to form the base set of tests.

In addition to the individual test cases themselves, detailed descriptions or samples of the actual expected output produced by each test case must also be supplied and maintained.

b. Information Output. The output from regression testing is simply the output produced by the software from the execution of each of the individual test cases.

c. Outline of Method. Regression testing is the process of retesting software in order to detect errors which may have been caused by program changes. The technique requires

the utilization of a set of test cases which have been developed (ideally, using functional testing sec. 4.3.2.3) to test all of the software's functional capabilities. If an absolute determination of those portions of the software that can potentially be affected by a given change can be made, then only those portions need to be tested. Associated with each test case is a description or sample of the correct output for that test case. When the tests have been executed, the actual output is compared with the expected output for correctness. As errors are detected during the actual operation of the software which were not detected by regression testing, a test case which could have uncovered the error should be constructed and included with the existing test cases.

Although not required, tools can be used to aid in performing regression testing. Automatic test harnesses can be used to assist the managing of test cases and in controlling the test execution. File comparators can often be useful in verifying actual output with expected output. Assertion processors are also useful in verifying the correctness of the output for a given test.

d. Example. An example of regression testing is given as follows.

Application. A transaction processing system contains a dynamic data field editor which provides a variety of input/output field editing capabilities. Each transaction is comprised of data fields as specified by a data element dictionary entry. The input and output edit routine used by each data field is specified by a fixed identifier contained in a data field descriptor in the dictionary entry. When a transaction is input, each field is edited by the appropriate input editor routine as specified in the dictionary entry. Output editing consists of utilizing output editor routines to format the output.

Error. An input edit routine to edit numeric data fields was modified to perform a fairly restrictive range check needed by a particular transaction program. Current system documentation indicated that this particular edit routine was only being used by that single transaction program. However, the documentation was not up-to-date in that another, highly critical, transaction program also used the routine, often with data falling outside of the range check needed by the other program.

Error discovery. Regression testing would uncover the error given that a sufficient set of functional tests were used for performing the testing. If only the transaction program for

which the modification was made was tested, the error would not have been discovered until actual operation.

e. Effectiveness. The effectiveness of the technique depends upon the quality of the data used for performing the regression testing. If functional testing is used to create the test data the effectiveness will be that of functional testing (highly effective). The burden and expense associated with the technique, particularly for small changes, can appear to be prohibitive. It is, however, often quite straightforward to determine which functions can be potentially affected by a given change. In such cases, the extent of the testing can be reduced to a more tractable size.

f. Applicability. This method is applicable during the unit and module test phases.

g. Maturity. Regression testing is a mature method. It is widely used.

h. User Training. No special training is required in order to apply this technique. If tools are used in support of regression testing, however, knowledge of their use will be required. Moreover, successful application of the technique will require establishment of procedures and the management control necessary to ensure adherence to those procedures.

i. Costs. Since testing is required as a result of software modifications anyway, no additional burden need result because of the method (assuming that only the necessary functional capabilities are retested). The use of tools, however, to support it could increase the cost but it would also increase its effectiveness.

j. References.
(PAN 78)

4.6 BIBLIOGRAPHY

(ADA79) "Ada Environment Workshop." Sponsored by DoD High Order Language Working Group, November 27-29, 1979, San Diego.

(ADR81) Adrion, W. R., M. A. Branstad, and J. C. Cherniavsky. "Validation, Verification and Testing of Computer Software." NBS Special Publication 500-75, Superintendent of Documents, U.S. Documents, U.S. Govt. Printing Office, Wash, D.C., February 1981.

(AHO74) Aho, A.V., J.E. Hopcroft, and J.D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

(AHO77) Aho, A.V. and J.D. Ullman. Principles of Compiler Design. Addison-Wesley, 1977.

(AIR78) "Airborne Systems Software Acquisition Engineering Guidebook for Verification, Validation, and Certification", Air Force document, ASD-TR-79-5028, 1978.

(ALF76) Alford, M. W. "A Requirements Engineering Methodology for Real-Time Processing Requirements," TRW Software Series, TRW-SS-76-07, Systems Engineering and Integration Division, Sept., 1976.

(ALF79) Alford, M. A. "Theoretical Foundations of Toolsmithing." Proceedings of COMPSAC 79, IEEE Catalog No. 79CH1515-6C, November 1979.

(ALL76) Allen, F.E. and J. Cocke. "A Program Data Flow Analysis Procedure", CACM, Vol. 19, No. 3, March 1976.

(ARE80) "A Review of Software Maintenance Technology." (RADC TR-80-13) dtd Feb 80. Available from the U.S. Dept. of Commerce, National Technical Information service, 5285 Port Royal Road, Springfield, Virginia, 22151, Accession No. A083-985.

(AUT77) "Automated Data Systems Documentation Standards." Department of Defense standard 7935.1-S, September 1977.

(AUT79) "Automated Software Tools Catalog." Boeing Computer Services document 10236, 1979.

(BAR78) Barbuto, P., Jr. and J. Geller. "Tools for Top-Down Testing." Datamation, Vol. 24, No. 10, October 1978, pp. 178-182.

(BAR80) Barry, M. "Airborne Systems Software Acquisition Engineering Guidebook for Software Testing and Evaluation." TRW report ASD-TR-80-5023, March 1980.

(BEL74) Bell, D.E. and J.E. Sullivan. "Further Investigations into the Complexity of Software". (Tech. Rep. MTR-2874). Bedford, MA: MITRE. 1974.

(BEN78) Benson, J.P. and S.H. Saib. "A Software Quality Assurance Experiment", Proceedings of the Software Quality and Assurance Workshop, San Diego, Nov. 1978.

(BEN79) Bentley, J.L. "An Introduction to Algorithm Design", Computer, Feb. 1979.

(BLA71) Blair, J. "Extendable Non-Interactive Debugging." Debugging Techniques In Large Systems, Prentice-Hall, Englewood Cliffs, N.J., 1971, pp. 93-115.

(BOE75) Boehm, B., R. McClean, and D. Urfrig. "Some Experience with Automated Aides to the Design of Large-scale Reliable Software", IEEE Transactions of Software Engineering, SE-1, 1975(125-133).

(BOY75) Boyer, R.S., B. Elspas, and K.N. Levitt. "SELECT-A Formal System for Testing and Debugging Programs by Symbolic Execution", Proceedings of the International Conference on Reliability of Software, April 1975.

(BRA75) Bratman, H. and T. Court. "The Software Factory." Computer, May 1975, pp. 28-37.

(BRA77) Bratman, H. and M. C. Finfer. "Software Acquisition Management Guidebook: Verification." System Development Corporation report ESD-TR-77-263, August 1977.

(BRO75) Brooks. The Mythical Man-Month. Addison-Wesley, 1975.

(BRO77) Brown. "Impact of Modern Programming Practices on System Development." RADC-TR-77-121, 1977.

(BRO78) Brown, J.R. and K. Fischer. "A Graph Theoretic Approach to the Verification of Program Structures", Proceedings of the 3rd International Conference on Software Engineering, May 1978.

(BRO82) Brownell, L. "Jovial J73 Code Auditor", Technical Report, Proprietory Software Systems, Inc., 9911 W. Pico Blvd., Los Angeles, CA 90035, 16 March 1982.

(BRO83) Brown, P.J. "Error Messages: The Neglected Area of Man/Machine Interface?", CACM, Vol. 26, No.4, April 1983.

(BUL74) Bulut, N. and M.H. Halstead. "Impurities Found in Algorithm Implementation", SIGPLAN Notices, 1974.

(BUX80) Buxton J. N. "An Informal Bibliography on Programming Support Environments." SIGPLAN Notices, December 1980.

(CAM81) Campbell, O. and S. Saib. "Embedded Software Verification Through Instrumentation", NAECON 81, May 19-21, 1981, Vol. I, pp. 395-401.

(CHE79) Cheatham, T.E., G.H. Holloway, and J.A. Townley. "Symbolic Evaluation and the Analysis of Programs", IEEE Transaction on Software Engineering, SE-5,4, July 1979.

(CHO) Chow, T.S. "A Generalized Assertion Language", Proceedings 2nd ICSE, S.F., Calif., pp. 392-399.

(CLA76) Clarke, L.A. "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transaction of Software Engineering, SE-2, Sept. 1976.

(CLA78) Clarke L. "Top-Down Testing with Symbolic Execution." Digest for the Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Florida, 18-20 December 1978, pp. 191-196.

(COC70) Cocke, J. and T.J. Schwartz. Programming Languages and Their Compilers, Preliminary Notes, Second Revised Version, Courant Institute of Mathematical Sciences, New York, 1970.

(COD76) "Code Reading Structured Walk-Throughs and Inspections", IBM, IPTO, Support Group, World Trade System Center, Postbos 60, Zoetenmeer, Netherlands, March 1976.

(CON70) "Control Flow Analysis". SIGPLAN Notices, 1970, pp. 1-19.

(COR76) Cornall, L.M. and M.H. Halstead. "Predicting the Number of Bugs Expected In a Program Module", (Tech. Rep. CSD-TR-205). West Lafayette, IN: Purdue University, Computer Science Department, October 1976.

(CRO75) Crocker, S. and B. Balzer. "The National Software Works: A New Distribution System for Software Development Tools." Workshop on Currently Available Testing Tools, April 1975, p. 21.

(CUR79) Curtis, B., S.B. Sheppard, and P. Milliman. "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics", Proceedings of the Fourth International Conference on Software Engineering. New York: IEEE, 1979.

(DAL77) Daly, E.B. "Management of Software Development", IEEE Transactions on Software Engineering, May 1977.

(DAV) Davis, C.G. "Testing Large, Real-Time Software Systems", Infotech State-of-the-Art Report - Software Testing, Infotech International, Berkshire, England,Vol. 2, pp. 85 - 105.

(DEC78) DEC IAS/RSX-11 "Utilities Procedure Manual", Digital Equipment Corporation, 1978.

(DEF79) "Defense Mapping Agency: Modern Programming Environment Study-Final Technical Report." Boeing Computer Services and Planning Systems International, Contract No. SB 1438(A)-79-C-001, October 1979.

(DEM79) DeMillo, R.A., R.J. Lipton, and F.G. Sayward. "Program Mutation: A New Approach to Program Testing", Infotech State-of-the-Art Report on Software Testing, V.2, INFOTECH/SRA, 1979, pp.107-127.

(DEM83) DeMillo, R. A., and R. J. Martin. "The Software Test and Evaluation Project: A Progress Report." Proceedings of the National Conference on Software Test and Evaluation, 1-3 February 1983.

(DER76) DeRemer and Kron. "Programming-in-the-Large Versus Programming-in-the-Small." IEEE Transactions on Software Engineering, June 1976.

(DEU81) Deutsch, M.S. "Software Project Verification and Validation", IEEE Computer, April 1981.

(DON80) Donahoo, J. D. and D. Swearinger. "A Review of Software Maintenance Technology." RADC report RADC-TR-80-13, February 1980.

(ELM73) Elmendorf, W.R. "Cause-Effect Graphs in Functional Testing", TR-00.2487, IBM Systems Development Division, Poughkeepsie, New York, 1973.

(ELS76) Elshoff, J.L. "Measuring Commercial PL/1 Programs Using Halstead's Criteria", SIGPLAN Notices, 1976,11, 38-46.

(ELS72) Elspas, B., et. al. "An Assessment of Techniques for Proving Program Correctness", ACM Computing Surveys, 4, June 1972.

(END75) Endres. "An Analysis of Errors and Their Causes in System Programs." IEEE Transactions on Software Engineering, June 1975.

(FAC77) "Factors in Software Quality", Final report, RADC-TR-77-369, 1977.

(FAG76) Fagan, M.E. "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, No. 3, 1976.

(FAI78) Fairley, R. E. "Tutorial: Static Analysis and Dynamic Testing of Computer Software." Computer, April 1978, pp. 14-24.

(FEL79) Feldman. "MAKE - A Program for Maintaining Computer Programs." Software Practice and Experience, April 1979.

(FIS74) Fischer, K.F. "User's Manual for Code Auditor, Code Optimizer Advisor, Unit Consistency Analyses", TRW Systems Group, Redondo Beach, Calif., July 1974.

(FIT78) Fitzsimmons, A.B. and L.T. Love. "A Review and Evaluation Science". ACM Computing Surveys, 1978, 10, 13-18.

(FLE79) Fleiss, J., G. Phillips, and A. Alvarez. "Compiler Acceptance Guidebook." RADC report RADC-TR-77-148, May 1979.

(FLO67) Floyd, R.W., T.J. Schwartz, editor. "Assigning Meanings to Programs", Mathematical Aspects of Computer Science, 19, American Mathematical Society, Providence, R.I., 1967.

(FOS76) Fosdick, L.D. and L.J. Osterweil. "Data Flow Analysis in Software Reliability", ACM Computing Surveys, 8, pp.305-330, Sept. 1976.

(FRE77) Freedman, D.P. and G.M. Weinberg. Ethno-Technical Review Handbook. Ethnotech, Inc., 1977.

(FUM76) Fumani, Y. and M.H. Halstead. "A Software Physics Analysis of Akiyama's Debugging Data", Proceedings of the MRI 24th International Software Engineering. New York: Polytechnic Press, 1976.

(GAN79) Gannon, C., R.N. Meeson, and N.B. Brooks. "An Experimental Evaluation of Software Testing - Final Report," General Research Corp., CR-1-854, sponsored by Air Force Office of Scientific Research, May 1979.

(GAN80) Gannon, C. "Jovial J73 Automated Verification System-Study Phase", RADC-TR-80-261, August 1980, NTIS accession no. A091-190.

(GIL77) T. Gilb, "Software Metrics," Winthrop Publishers, Inc., Cambridge, Mass, 1977.

(GLA76) Glass, R. L. "An Experiment in the Use of Analyzers as a Computer Software Reliability Tool in the BAC Project Environment", Boeing Aerospace Company D180-19987-1, August 1976.

(GLA78) Glass, R. L. "Software Reliability Methodology Survey and Guidebook", The Boeing Company, D180-22930-1, 1978.

(GLA79A) Glass, R. L. Software Reliability Guidebook. Prentice-Hall, 1979.

(GLA79B) Glass, R. L. "Software Reliability at Boeing Aerospace: Some New Findings." *Boeing Aerospace Co. report* D180-25392-1, September 1979.

(GLA79C) Glass, R. L. "Real Time Software Debugging and Testing: Proposed Solutions." Boeing Aerospace Co. report D180-25249-3, September 1979.

(GLA81) Glass, R. L. and R. Noiseux. Software Maintenance Guidebook. Prentice Hall, Inc., 1981.

(GLA-A) Glass, R. L. "Automated Tools for Software IV & V." The Boeing Co. Unpublished draft.

(GLA-B) Glass, R. L. "Recommended: A Minimum Standard Software Toolset". The Boeing Co., Unpublished draft.

(GOD77) Godoy and Engels. "Software Sneak Analysis." Proceedings of the AIAA Conference on Computers in Aerospace, 1977.

(GOE81) Goel, Dr. Amrit. "A Guidebook for Software Reliability Assessment", prepared under contract No. F30602-81-C-0169, for RADC.

(GOO75) Good, D.I., R.L. London, and W.W. Bledsoe. "An Interactive Program Verification System", Proceedings of the 1975 International Conference on Reliable Software, IEEE Catalog #75CH0940-7CSR.

(GOR76) Gordon, R.D. and M.H. Halstead. "An Experiment Comparing FORTRAN Programming Times with the Software Physics Hypothesis", AFIPS Proceedings, 1076, 45, 935-937.

(HAL73) Halstead, M.H. "An Experimental Determination of the 'Purity' of a Trivial Algorithm", ACM SIGME Performance Evaluation Review, 1973, 2(1), 10-15.

(HAL77) Halstead, M.H. Elements of Software Science, Elsevier Computer Science Library, 1977.

(HAM82) Hamer, P.G., G.D. Frewin. "M. H. Halstead's Software Science-A Critical Examination", Proceedings of the 6th International Conference on Software Engineering, Tokyo, Japan, September, 1982.

(HAR71) Hartwic, R.D. "The Advanced Targeting Study", SAMSO-TR-71-124, Vol.1, June 1971.

(HEC73) Hecht, M.S. and J.D. Ullman. "Analysis for a simple algorithm for global data flow problems". Proc. ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages. Boston, Mass., Oct. 1973.

(HEC75) Hecht, M.S. and J.D. Ullman. "A Simple Algorithm for Global Data Flow Analysis Problems", Siam Journal of Computing, Vol. 4, No. 4, December 1975.

(HEC81A) Hecht, H. "Synopsis of Interviews from a Survey of Software A Tool Usage." SoHaR Inc. report NBSIR 81-2388, November 1981.

(HEC81B) Hecht, H. "Final Report: A Survey of Tool Usage." NBS Special Publication 500-82, Superintendent of Documents, U.S. Govt. Printing Office, Wash, D.C. 20402, November 1981.

(HEC82) Hecht, H. "The Introduction of Software Tools." NBS Special Publication 500-91, Superintendent of Documents, U.S. Govt. Printing Office, Wash, D.C. 20402, September 1982.

(HEI82) Heidler, W., et al. "Software Testing Measures." RADC-TR-82-135, May 1982.

(HEN81) Henry, Sallie, et.al. On the Relationships Among Three Software Metrics, ACM Sigmetrics, Perf. Eval. Rev., 1981.

(HET73) Hetzel, W. Program Test Methods. Prentice-Hall, 1973.

(HET76) Hetzel, W. "An Experimental Analysis of Program Verification Methods", Ph.D. Thesis, University of North Carolina, 1976.

(HOA61) Hoare, C.A.R. "Partition (Algorithm 63) and QUICKSORT (Algorithm 64)", CACM, Vol.4, No. 7, July 1961.

(HOA64) Hoare, C.A.R. "QUICKSORT", Computer Journal, Vol.5, No.1, 1964.

(HOA71) Hoare, C.A.R. "Proof of a Program: FIND", CACM, Vol.14,No.1, Jan. 1971, pp.39-45.

(HOR75) Horowitz. Practical Strategies for Developing Large Software Systems. Addison-Wesley, 1975.

(HOR78) Horowitz, E. and S. Sahni. Fundamentals of Computer Algorithms. Computer Science Press, Potamac, MD. 1978.

(HOU81A) Houghton, R. C., Jr. "Features of Software Development Tools." NBS Special Publication 500-74, Superintendent of Documents, U.S. Govt. Printing Office, Wash, D.C. 20402, February 1981.

(HOU81B) Houghton, R. C., Jr., editor. "Proceedings of NBS/IEEE/ACM Software Tool Fair." NBS Special Publication 500-80, Superintendent of Documents, U.S. Govt. Printing Office, Wash, D.C. 20402, October 1981.

(HOU82) Houghton, R. C., Jr. "Software Development Tools." NBS Special Publication 500-88, Superintendent of Documents, U.S. Govt. Printing Office, Wash, D.C. 20402, March 1982.

(HOW A) Howden, W. E. "Functional Testing and Design Abstractions". A Journal of Systems and Software (to appear).

(HOW) Howden, W. E. "Symbolic Testing - Design Techniques, Costs, and Effectiveness", U.S. Dept. of Commerce, NTIS PB-268, 517, Springfield, VA.

(HOW75) Howden, W. E. "Methodology for Generation of Program Test Data", IEEE Transactions on Computers, TC-24, May,1975.

(HOW77) Howden, W.E. "Symbolic Testing and the Dissect Symbolic Evaluation System", IEEE Trans. on Software Engineering, Vol. SE-3, No. 4, July 1977.

(HOW78A) Howden, W. E. "An Evaluation of the Effectiveness of Symbolic Testing." Software Practice and Experience, July 1978.

(HOW78B) Howden, W. E. "Selection of Fortran Static Analysis Techniques." University of Victoria report DM-147-IR, August 1978.

(HOW78C) Howden, W. E. "Functional Program Testing." University of Victoria report DM-146-IR, August 1978.

(HOW79) Howden, W. E. "An Analysis of Software Validation Techniques for Scientific Programs." University of Victoria report DM-171-IR, March 1979.

(HOW80A) Howden, W. E. "Validation of Scientific Programs", U.S. National Bureau of Standards, Wash., D.C., 1980.

(HOW80B) Howden, W. E. "Completeness Criteria for Testing Elementary Program Functions", University of Victoria, Department of Mathematics, May 1980.

(HOW80C) Howden, W. E. "Functional Program Testing". IEEE Transactions on Software Engineering, SE-7, March 1980.

(HOW80D) Howden, W. E. "Functional Testing and Design Abstractions". Journal of Systems and Software, Vol. 1, 307-313, 1980.

(HUA75) Huang. "An Approach to Program Testing", ACM Computing Surveys, September 1975.

(HUA79) Huang, J.C. "Detection of Data Flow Anomaly Through Program Instrumentation", IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979.

(JAC71) Jackson and Bravdica. "Software Validation of the Titan IIIC Digital Flight Control System Using a Hybrid Computer." Proceedings of the 1971 Fall Joint Computer Conference.

(KAR78) Karr, M., D.B. Loveman, III. "Incorporation of Units into Programming Languages", CACM, Vol.21, No.5, pp.385-391, May 1978.

(KEN76) Kennedy, Ken. "A Comparison of Two Algorithms for Global Data Flow Analysis", Siam Journal of Computing, Vol. 5, No. 1, March 1976.

(KER76) Kernighan B. W. and P. J. Plauger. Software Tools. Addison-Wesley, 1976.

(KIL73) Kildall, G.A. "A unified approach to global program optimization". Proc. ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages. Boston, Mass., Oct. 1973.

(KIN76) King, J.C. "Symbolic Execution and Program Testing", CACM, 19, 7, July 1976, pp.385-394.

(KIN81) King, Bill. "ARGUS on BITS." Boeing Computer Services-SAMA Software Engineering Technology, November 1981.

(LAP74) L. J. LaPadula, "Engineering of Quality Software Systems, Volume VIII, Software Reliability Modeling and Measurement Techniques," RADC-TR-74-325, Mitre Corp., Bedford, Mass., 1975 (NTIS AS/A-007773).

(LIP78) Lipton, R.J. and F.G. Sayward. "The Status of Research on Program Mutation", Digest of the Workshop on Software Testing and Test Documentation, Fort Lauderdale, FLA., 1978, pp.355-373.

(LOV76) Love, L.T. and A. Bowman. "An Independent Test of the Theory of Software Physics", SIGPLAN Notices, 1976,11 pp.42-49.

(MAN74) Mangold, E.R. "Software Error Analysis and Software Policy Implications", IEEE EASCON, 1974, pp. 123-127.

(MAN78) Manna, Z. and R. Walding. "The Logic of Computer Programming", IEEE-TSE, SE-4, No.3, May 1978, pp.199-229 (especially pages 199-204).

(MAR78) "Problem Program Evaluator (PPE) User Guide", Boole and Baggage, Inc., Sunnyvale, Calif., March 1978.

(MCC76) McCabe, T.J. "A Complexity Measure", IEEE Transaction on Software Engineering, Vol. SE-2, No.4, December 1976.

(MEL79) Melton, R. "Fortran Automated Verification System (FAVS)", Vol. I, technical report, RADC-TR-78-268, Jan. 1979, NTIS accession no. A065-405.

(MEL81) Melton, R. "Cobol Automated Verification System - Study Phase", RADC-TR-81-11, March 1981, NTIS accession no. A098-755.

(MER81) Merilatt, R. L., M. K. Smith, and L. L. Tripp. "Computer Software Verification and Validation: A General Guideline." Boeing Computer Services report BCS-40342, June 1981.

(MEY75) Meyers, G. Reliable Software Through Composite Design, Petrocelli/Charter, 1975.

(MIL72) H. D. Mills, "On the Statistical Validation of Computer Programs, FSC-72-6105, IBM Federal Systems Division, Gaithersburg, Md., 1972.

(MIL75) Miller, E. and R.A. Melton. "Automated Generation of Testcase Datasets", 1975 International Conference on Reliable Software, Los Angeles, April 1975.

(MIL81) Miller, E. and W. E. Howden. TUTORIAL: Software Testing and Validation Techniques. IEEE Computer Society Press, 1981.

(MYE76) Myers, G. Software Reliability: Principles and Practices. Wiley-Interscience, New York, 1976.

(MYE78) Myers. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections." Communications of the ACM, September 1978.

(MYE79) Myers, G. The Art of Software Testing. Wiley-Interscience Publication, 1979.

(NAF72) Naftaly, S.M. and Cohen, M.C. "Test Data Generators and Debugging Systems", Workable Quality Control, Part I and II, Data Processing Digest, Vol.18, No.2 and 3, February and March, 1972.

(NBS80) "NBS Software Tools Database." dtd Oct 80. Available from the U. S. Dept. of Commerce, National Technical Information Service, 5285 Port Royal Road, Springfield, Virginia, 22151, Accession No. PB81-124935.

(NG78) Ng and Young. "A 1900 Fortran Post Mortem Dump System." Software Practice and Experience, July 1978.

(OST76) Osterweil, L.J. and L.D. Fosdick. "DAVE-A Validation Error Detection and Documentation System for Fortran Programs", Software Practice and Experience, 6, pp.473-486, Sept. 1976.

(OST78) Osterweil, L. J., J. R. Brown, and L. G. Stucki. "ASSET: A Lifecycle Verification and Visibility System." Proceedings of COMPSAC 78, IEEE Catalog No. 78CH1338-3C, November 1978, pp. 30-35.

(OTT79) Ottenstein, L.M. "Quantitative Estimates of Debugging Requirements", IEEE Transactions on Software Engineering, 1979, Vol.5, pp.504-514.

(PAI77) Paige. "Software Testing Principles and Practice Using a Testing Coverage Analyzer." Transactions of the Software '77 Conference, October 1977.

(PAN78) Panzl,D.J. "Automatic Software Test Drivers", IEEE Computer, April 1978.

(POW82A) Powell, P. B., editor. "Software Validation, Verification, and Testing Technique and Tool Reference Guide." NBS Special Publication 500-93, Superintendent of Documents, U.S. Govt. Printing Office, Wash, D.C. 20402, September 1982.

(POW82B) Powell, P. B., editor. "Planning for Software Validation, Verification, and Testing." NBS Special Publication 500-98, Superintendent of Documents, U.S. Govt. Printing Office, Wash, D.C. 20402, November 1982.

(PRE79) "Preliminary Ada Reference Manual", SIGPLAN Notices, Vol.14, No.6, part A, June 1979.

(PRO83) "Proceedings of the National Conference on Software Test and Evaluation." National Security Industrial Association, Software Group, 1-3 February 1983.

(QUA79) "Quantitative Software Models." Data and Analysis Center for Software, order No. SRR-1, RADC/ISISI (315) 336-0937, Autovon 587-3395.

(QUA83A) "Quality Metrics for Distributed Systems", Final report, Boeing Aerospace Company document, D-182-11377-1, -2, -3, 1983.

(QUA83B) "Quality Metrics Framework Enhancements for Software Aquisition" (CDRL A003), RADC contract F30602-82-C-0137 with Boeing Aerospace Company, July 1983.

(RAM75) Ramamoorthy, C.V. and K.H. Kim. "Software Monitors Aiding Systematic Testing and Their Optional Placement", Proceedings of the First National Conference on Software Engineering, IEEE Catalog No. 75CH0992-8C, September, 1975.

(REI74) Reifer, D. J. and R. L. Ettenger. "Test Tools: Are They A Cure-All?". SAMSO-TR-75-13, October 1974.

(REI77) Reifer, D. J. and Trattner. "A Glossary of Software Tools and Techniques." IEEE Computer, July 1977.

(REI80) Reifer, D. J. and H. A. Montgomery. "Final Report, Software Tool Taxonomy." SMC-TR-004, 1 June 1980.

(REI) Reifer, D. J. "Software Tools Directory." Reifer Consultants Inc. 2733 Pacific Coast Highway, Suite 203, Torrance, CA 90505.

(RIC81) Richardson, D.J., L.A. Clarke. "A Partition Analysis Method to Increase Program Reliability". Proceedings of the Fifth International Conference of Software Engineering, 1981, pp.244-253.

(RUD77) B. Rudner, "Seeding/Tagging Estimation of Software Errors: Models and Estimates," RADC-TR-77-15, Polytechnic Institute of New York, 1977 (NTIS AD/A-036655).

(RYD75) Ryder, B.G.and A.D. Hall. "The PFORT Verifier", Computing Science Technical Report #12, Bell Laboratories, Murry Hill, New Jersey, March 1975.

(SAI82) Saib, S. H., et al. "Validation of Real-Time Software for Nuclear Plant Safety Applications." Electric Power Resear·h Institute report EPRI NP-2646, Project 961 Final Report, November 1982.

(SCH73) Schaefer, M., A Mathematical Theory of Global Program Optimization. Prentice-Hall, Englewood Cliffs, N.J., 1973.

(SCH81) Schindler, M. "Today's Software Tools Point to Tomorrow's Tool Systems." Electronic Design, 23 July 1981, pp. 73-110.

(SCH79) Schneidewind and Hoffman. "An Experiment in Software Error Data Collection and Analysis." Transactions on Software Engineering, May 1979.

(SHE83) Shen, V.Y., et.al. Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support, IEEE Trans. on SW Eng., March 1983.

(SHN80) Shneiderman, B. Software Psychology-Human Factors in Computer and Information Systems. Winthrop Publishing, 1980.

(SMI76) Smith, P. "Fortran Code Auditor Users' Manual", RADC-TR-76-395, Vol. I, December 1976, NTIS accession no. A035-778.

(SMI79) Smith, C.U. and Browne, J.C. "Performance Specifications and Analysis of Software Designs", Proc. Conference on Simulation, Measurement and Modeling of Computer Systems, Boulder, CO., August 1979.

(SMI80) Smith, C.U. "The Prediction and Evaluation of the Performance of Software from Extended Design Specification", Ph.D. Dissertation, University of Texas at Austin, August 1980.

(SMI81) Smith, M. K. and D. R. Hudson, et al. "A Report on a Survey of Validation and Verification Standards and Practices at Selected Sites." Boeing Computer Services report BCS-40345, June 1981.

(SOF77) "Software Acquisition Management Guidebook: Validation and Certification", Air Force document, ESD-TR-77-326, 1977.

(SOF80) "Software Quality Metrics Enhancements", Final report, RADC-TR-80-109, 1980.

(SOF82) "Software Engineering Automated Tools Index." Software Research Associates, 1982.

(SOF83) "Software Interoperability and Reusability", Final report, Boeing Aerospace Company document, D182-11340-1, -2, 1983.

(SPE79) "Sperry Univac Series 1100 Fortran (ASCII) Programmer Reference," Sperry Rand Corporation, 1979.

(SPE82) "Specification of Software Quality Attributes", Interim reports, Boeing Aerospace Company documents, D182-11310-1, D182-11378-1, 1982.

(STA77) Stanfield and Skrukrud. "Software Aquisition Management Guidebook Software Maintenance Volume," System Development Corp., TM-5772/004/02, November 1977.

(STU73) Stucki, L.G. "Automatic Generation of Self-Metric Software", Proc. 1973 IEEE Symposium on Computer Software Reliability, 94(1973).

(STU75) Stucki, L. G. and G.L. Foshee. "New Assertion Concepts for Self-Metric Software", Proc. 1975 Conference on Reliable Software, pp.59-71.

(STU81) Stucki, Leon G. "Using ARGUS on EKS II." Boeing Computer Services, March 1981.

(SUK77) Sukert. "A Multi-Project Comparison of Software Reliability Models." Proceedings of the AIAA Conference on Computers in Aerospace, 1977.

(SUM) "Summary of Software Testing Measures." Software Research Associates report SRA TN-843.

(SYS77) "Systematic Software Development and Maintenance (SSDM)", Boeing Computer Services Document #10155, February 1977.

(TAY79) Taylor, R. N., R. L. Merilatt, and L. J. Osterweil. "Integrated Testing and Verification System for Research Flight Software." Boeing Computer Services report NAS1-15253, July 1979.

(TAY80) Taylor, R.N. "Assertions in Programming Languages", SIGPLAN Notices, Vol.15, No.1, January 1980, pp.105-114.

(TAY80B) Taylor, R.N. and L.J. Osteweil. "Anomaly Detection in Concurrent Software by Static Data Flow Analysis", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, pp. 265-278, May 1980.

(TEI72) Teichroew, D. "A Survey of Languages for Stating' Requirements for Computer-Based Information Systems", the University of Michigan, Proceedings of the Fall Joint Computer Conference, 1972, pp.1203-1224.

(TEI77) Teichroew, D. and E.A. Hershey, III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, SE-3, 1977, (41-48).

(THA76) Thayer, T.A., et al. "Software Reliability Study." RADC report RADC-TR-76-238, August 1976.

(THR75) "THREADS: A Functional Approach to Project Control", Computer Sciences Corporation, El Segundo, CA, 1975.

(ULL73) Ullman, J.D. "Fast Algorithms for the elimination of common subexpressions". Acta Informatica, 2(1973), pp. 191-213.

(WEA78) "Weapon System Software Development." Military Standard MIL-STD-1679 (NAVY), 1 December 1978.

(WEI71) Weinberg, G.M. "Programming as a Social Activity", The Psychology of Computer Programming. Van Nostrand, Reinhold, 1971.

(WEI77) Weide, B. "A Survey of Analysis Techniques for Discrete Algorithm", Computing Surveys, Vol.9, No.4, Dec. 1977.

(WEI78) Weiss. "Evaluating Software Development by Error Analysis." Naval Research Lab NRL-8268, December 1978.

(WES79) Western District Utilities Manual, Boeing Computer Services Document #G0031 Rev. A, June 1979.

(WHI80) White, L.J. and E.I. Cohen. "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No.3, May 1980.

(WIN79) Winograd. "Beyond Programming Languages." Communication of the ACM, July 1979.

(WOO79) Woodfield. "An Experiment on Unit Increase in Program Complexity". IEEE Transactions on Software Engineering, March 1979.

(YEH77) Yeh, R.T., editor. Current Trends in Programming Methodology, Volume II. Prentice-Hall, Inc., 1977.

(YOU77) Yourdon, E. Structured Walk-throughs. Yourdon, Inc. 1977.

5.0 ACQUISITION LIFE CYCLE

5.1 AIR FORCE PHASED ACQUISITION

This section contains descriptions of Air Force phased acquisition objectives for development test and evaluation (DT&E), operational test and evaluation (OT&E), initial operational test and evaluation (IOT&E), follow-on operational test and evaluation (FOT&E), and software verification and validation (V&V). Figure 5.1-1 shows the relationship between the test phases used in this handbook, the Air Force software life cycle and the test and evaluation phases. Reference figure 2-10 for definitions of the nine test phases used in this guidebook.

5.1.1 Test and Evaluation During the Acquisition Process

There are two kinds of test and evaluation (T&E) in the system acquisition process: DT&E and OT&E. Either may occur at any point in the life cycle of the system, subsystem, or item of equipment (all hereafter referred to as a system). Their primary purposes are to identify, assess, and reduce the acquisition risks, to evaluate operational effectiveness and operational suitability, and to identify any deficiencies in the system. Adequate T&E must be performed before each major decision point to make sure that the major objectives of one phase of the system acquisition life cycle have been met before the next phase is begun. Quantitative data must be used to the maximum extent practical, to show that the major objectives have been met. Subjective judgment, relative to systems performance, must be minimized.

The following two sections describe DT&E and OT&E objectives and the relationship of DT&E and OT&E with the test phases used in the handbook. Detailed information on T&E can be found in AFR 80-14, paragraphs 19 through 22.

5.1.1.1 Development Test and Evaluation

Through DT&E, the Air Force must demonstrate that (1) the system engineering design and development are complete, (2) design risks have been minimized, and (3) the system will perform as required and specified. It involves an engineering analysis of the system's performance, including its limitations and safe operating parameters. The system design

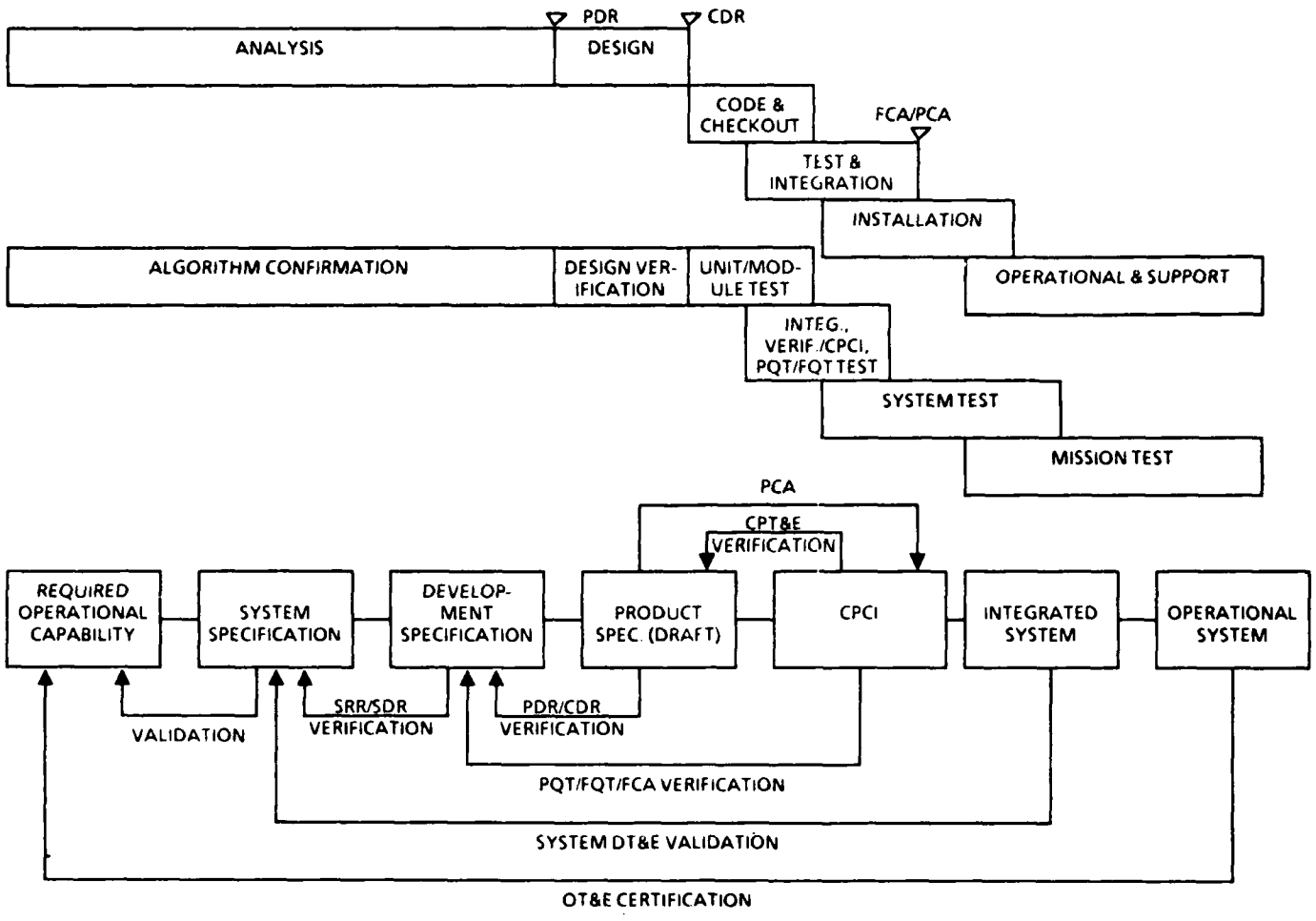


Figure 5.1-1. The Scope of Verification, Validation, and Certification and Software Life Cycle Phases vs. Test Phases

is tested and evaluated against engineering and performance criteria specified by the implementing command. DT&E addresses the logistic engineering aspects of the system design and may go on all through the life cycle. It may include testing not completed before the first major production decision. It may also involve testing product improvements or modifications designed to correct identified deficiencies or to reduce life cycle costs.

The types of testing that occur during DT&E according to the test phases used in this handbook are algorithm confirmation testing through system testing. In other words, DT&E activities include very low level testing to high level testing. The testing objectives include verifying that the algorithm will satisfy the requirements imposed on the software design, verifying that the design is a correct implementation of the specified requirements, verifying that the unit's logic and interfaces satisfies the design specifications, verifying that the computer program configuration item (CPCI) is a correct implementation of the specified design, verifying that all specified real-time and functional requirements are satisfied, and verifying that the system is in agreement with the system level specifications.

5.1.1.2 Operational Test and Evaluation

OT&E is conducted, in conditions made as realistic as possible, throughout the system life cycle. It is done to estimate (or to refine estimates of) a system's operational effectiveness and operational suitability in order to identify any operational deficiencies and the need for any modifications.

Through OT&E, the Air Force measures the system against the operational criteria outlined in pertinent program documentation (e.g., system operational concepts) developed by DOD, HQ USAF, and using and supporting commands. Information is provided on organizational structure, personnel requirements, doctrine, and tactics.

OT&E is used to provide data to verify operating instructions, computer documentation, training programs, publications, and handbooks. It uses personnel with the same skills and qualifications as those who will operate, maintain, and support the system when deployed.

Types of OT&E include initial OT&E (IOT&E) and follow-on OT&E (FOT&E). On certain programs, qualification OT&E (QOT&E) is conducted instead of IOT&E. For guidance on conducting OT&E, see AFM 55-43.

IOT&E is conducted before the first major production decision. It is done by the OT&E command or agency (hereafter called OT&E command) designated by HQ USAF. As a rule, it is done using a prototype, preproduction article or a pilot production item as the test vehicle.

FOT&E is that operational testing usually conducted after the first major production decision or after the first production article has been accepted. It may go on all through the remainder of the system life cycle. In this case, it may be done to refine estimates of operational effectiveness and suitability, to identify operational deficiencies, to evaluate system changes, or to reevaluate the system against changing operational needs.

The types of testing that occur during OT&E according to the test phases used in this handbook are system and mission testing. That is, OT&E activities include very high level testing. The testing objectives include verifying that the entire system meets its system level specifications and verifying that the entire system meets the requirements of the mission.

5.1.2 Computer Program Verification and Validation

This section describes computer program V&V and its relationship to the computer program life cycle. Only those phases of the life cycle that pertain to software testing are included in this discussion. The relationship between the computer program life cycle and the test phases used in this handbook are also given.

5.1.2.1 Design Phase

All models should be checked for logic and completeness. In some cases it might be desirable to independently rederive equations to insure correctness. A scientific simulation of the system may be produced; that is, the algorithms are coded in a higher

order language such as Fortran and executed on a general purpose computer. This type of simulation is used to develop algorithms and to check system interfaces. The outputs from such a simulation may be useful for later comparison with actual system outputs.

The type of testing that occurs during the design phase according to the test phases used in the handbook is design verification testing. The objective of this type of testing is to verify that the design is a correct implementation of the specified requirements.

5.1.2.2 Code and Checkout Phase

After a program has been coded, it must be reviewed to ensure that it agrees with program specifications. This can be accomplished by cross-checking the code itself with earlier specifications, flow charts, and so forth, or by running the code in a simulated computer environment. One way to accomplish this type of checkout is by desk-checking; that is, by manually going through the code and comparing it to the specifications. Another method is a correctness proof; that is, a mathematical proof that code performs exactly the functions given in the coding specifications and no others. Correctness proofs are only practical for relatively small routines.

The types of testing that occurs during the code and checkout phase according to the test phases used in the handbook are primarily unit test and module test. Integration test and verification/CPCI test may also occur at this time. The objectives of unit and module test phases are to detect discrepancies between the unit's logic and interfaces, and its design specifications, and to detect discrepancies between the module's logic and interfaces, and its design specifications, respectively. Objectives for integration test and verification/CPCI test can be found in section 5.2.

5.1.2.3 Test and Integration Phase

Several different types of simulation are used in the test and integration phase. In one simulation, the operational program is run on the actual operational computer, while another computer is used to simulate the inputs to the interfacing electronic equipment. Another simulation is similar to this one, except that some system equipment is used in addition to the operational computer. A further simulation uses system equipment and some live inputs; for example, the use of a radar set against actual aircraft in flight.

The types of testing that occur during the test and integration phase according to the test phases used in the handbook are primarily integration, verification/CPCI, and PQT/FQT testing. Module testing may occur early in the test and integration phase. The test phase objectives are to combine and test units and modules in order to check that the interfaces are defined correctly, to verify that the computer program configuration item (CPCI) is a correct implementation of the specified design, and to verify that the real-time and functional requirements are satisfied.

5.1.2.4 Operational and Support Phase

Whenever changes are made to equipment of computer programs, the operational program must be retested. Simulations are useful for retesting the system. Simulation tools and accompanying documentation should be identified in the computer resources integrated support plan (CRISP) and acquired at the same time the system is acquired.

The types of testing that occur during the operational and support phase according to the test phases used in the handbook is mission testing. The objective of this testing is to verify that the entire system correctly fulfills the intended mission.

6.0 SAMPLE REQUIREMENT PARAGRAPHS FOR STATEMENTS OF WORK

6.1 INTRODUCTION

This section provides sample statement of work paragraphs, which are examples of the various approaches that can be used to specify the use of software testing techniques with the aid of the Software Test Guidebook. The sample SOW paragraphs are written to allow various degrees of constraint in the imposition of testing technique requirements, from very tightly constrained to mild guidance. They may need to be modified or specific details added by the Air Force acquisition manager to fit the development environment of a particular situation or the contractual relationship being considered.

The information in parentheses (...) must be filled in by the acquisition manager. The SOW paragraphs require that "implementation details" be provided in the Software Test Plan. Data item descriptions (DID) for software test plans may not include provisions for this type of information. Therefore, the DID may need to be modified to provide a section for the newly required information.

For tightly constrained requirements (sections 6.2 and 6.3), reference the desired testing techniques in section 4.0 by name and section number.

6.2 TIGHTLY CONSTRAINED--DIRECT SPECIFICATION

The following paragraph can be used to specify specific techniques identified in the guidebook. The guidebook would thus serve as a guide to AF personnel for selecting testing techniques for all software products. The testing techniques for each software product, such as ground support software and inflight software, would be separately determined.

Tightly Constrained - Direct Specification

> **Developed computer programs shall be tested in accordance with techniques described in the Software Test Guidebook, RADC-TR-84-XX. The following testing techniques shall be used:**

Technique Name	Guidebook Sect. #	Phase(s)

The testing techniques, and necessary implementation details, shall be described in the Computer Program Test Plan (See CDRL).

6.3 TIGHTLY CONSTRAINED—SUBSET SPECIFICATION

The following paragraph can be used to specify a minimum set of techniques plus the use of the guidebook by the contractor to select additional testing techniques for the entire software project or an individual computer program.

Tightly Constrained - Subset Specification

Developed computer programs shall be tested in accordance with techniques described in the Software Test Guidebook, RADC-TR-84-XX. As a minimum, the following testing techniques shall be used:

Technique Name	Guidebook Sect. #	Phase(s)

Selection of additional techniques, utilizing a test confidence level (TCL) of (....) shall be performed in accordance with procedures described in Section 2.0 of the Software Test Guidebook. The resulting set of test techniques, and necessary implementation details, shall be described in the Computer Program Test Plan (See CDRL). Rationale for the selection of those techniques, from the candidate set identified by use of the Software Test Guidebook, shall be provided.

6.4 MODERATELY CONSTRAINED SPECIFICATION

The following paragraph can be used to specify the use of the guidebook for selecting software testing techniques. A test confidence level is the only predetermined factor. The paragraph does not constrain the developer to use specific techniques, nor does it identify specific phases in which the techniques are to be applied.

Moderately Constrained

> **Developed computer programs shall be tested in accordance with techniques described in the Software Test Guidebook, RADC-TR-84-XX. Selection of techniques, utilizing a test confidence level (TCL) of (....) shall be performed in accordance with procedures described in Section 2.0 of the Software Test Guidebook. The resulting set of test techniques, and necessary implementation details, shall be described in the Computer Program Test Plan (See CDRL). Rationale for the selection of those techniques, from the candidate set identified by use of the Software Test Guidebook, shall be provided.**

6.5 LOOSELY CONSTRAINED SPECIFICATION

The following paragraph can be used to allow the contractor extensive freedom in selecting software testing techniques.

Loosely Constrained

> **Developed computer programs shall be tested in accordance with techniques described in the Software Test Guidebook, RADC-TR-84-XX. Selection of techniques shall be in accordance with procedures described in Section 2.0 of the Software Test Guidebook. The resulting set of test techniques, and necessary implementation details, shall be described in the Computer Program Test Plan (See CDRL). Rationale for the selection of those techniques, from the candidate set identified by use of the Software Test Guidebook, shall be provided.**

INTRODUCTION TO APPENDICES

The information in the five appendices was derived from a survey of Air Force testing requirements in five mission areas. This survey was done as part of the contract for the preparation of this handbook. The survey gathered information on programming characteristics and the development/support environments of typical application software for the five Air Force mission areas. Within each mission area, differences in application software which impact the most appropriate testing/verification strategy were investigated. The survey forms were supplemented by visits to representative Air Force software sites.

APPENDIX A: ARMAMENT

The software testing procedures described in this appendix are based on a representative site devoted largely to this mission. Therefore, the procedures covered in this appendix may not include all aspects of the armament mission.

A1. ARMAMENT DIVISION

The Armament Division (AD) is a developing agency for tactical weapon systems, and particularly for threat, missile, and scoring systems. All embedded software systems development at AD is performed by contractors. The contractors are usually small, specialized, high-technology companies, but larger aerospace companies also contribute to the systems development. The software contained in these systems typically tends not to be critical, even though the systems themselves may be critical. The contractors design, develop, and test the software according to contractor-defined standards and under the general contractual-level supervision of Air Force personnel. Testing practices vary widely among the many contractors supporting the AD.

A2. AD MISSION

The primary mission area applicable to the Armament Division is armament. Secondary mission areas are aeronautical and missile/space, including avionics systems and air-to-air and air-to-ground missile systems.

A3. SOFTWARE DEVELOPMENT ENVIRONMENT

The most significant category of software is the operational software for embedded systems. The embedded systems developed at AD are generally found in three categories:

a. Threat systems. Defensive and offensive radar, targeting and tracking systems, and electronic countermeasures.
b. Missile systems. Air-to-air and air-to-ground missile systems.
c. Scoring systems. Proximity detectors for projectiles, used in aerial gunnery training.

The data processing and administrative software area was not surveyed; typically, these systems are in place and are not subject to extensive new development. Also, the embedded operational software systems are more germane and critical to the primary mission of the AD.

General Environment. Software development is conducted as a part of embedded systems development. System requirements are defined by Air Force personnel and then the systems and the software are designed, implemented, and tested by contractors selected in competitive bidding. The Air Force may participate in the definition and specification of detailed requirements. The companies range from small businesses to major aerospace corporations. Procurement requirements for software are similar for all systems development and their implementation is monitored by design reviews, audits, and detailed reviews of contractor-furnished documentation.

Software (computer programs and associated data) typically comprises a significant share of the contractors' system development effort (labor), ranging up to one-half or even more of the total effort.

Standards. The Armament Division complies with the 800-series Air Force regulation for systems development. The following regulations and standards are applicable to embedded systems and software development:

AFR 80-14 (Test and Evaluation).
AFR 800-14 (Management of Computer Resources in Systems).
MIL-STD-483 (Configuration Management Practices).
MIL-STD-1521A (Reviews and Audits).
MIL-STD-490 (Specification Practices).
MIL-STD-1750A (16-Bit Instruction Set Architecture).
MIL-STD-1589A (JOVIAL J73 Language).
MIL-STD-1815 (Ada Language).
IEEE STD-716 (ATLAS Language).
American National Standard X3.9 (FORTRAN 77 Language).

Administration. Contractors are required to identify and account for embedded software as computer program configuration items (CPCI), including support computer programs (such as ATE software). The programs are controlled using allocated configuration

identification and product configuration identification, with associated Part I and Part II specifications. CPCI-to-CPCI and CPCI-to-hardware interface specifications are also required. Computer programs are subjected to a sequence of reviews and audits: preliminary design review, critical design review, functional configuration audit, and physical configuration audit.

Languages. Approved high-order languages are mandated for embedded computer programs. The specified languages are JOVIAL J73, Ada, and FORTRAN 77, in that order of priority. The specified language for automatic test equipment (ATE) is ATLAS. Exceptions to the priority or the approved list must be authorized; the use of assembly language or nonapproved higher order language (HOL) subsegments must also be authorized.

Support Software. Contractors are encouraged to use off-the-shelf components and support tools. The following support tools are required:

a. An efficient compiler (in terms of code generated) and code generator for an approved HOL.
b. A software development station with aids, including a programmable read only memory (PROM) programmer, if applicable.
c. A complete support software library, including but not limited to an editor, linking loader, and run-time support routines.
d. Compatible hardware and software peripheral equipment.

Capacity Requirements. Embedded software is required to have a 30% spare capacity in memory utilization. Also, the software is required to exercise only 70% of the computer's throughput and input/output channel capacity.

Coding Standards. Contractors must establish coding standards for software development. The following minimum requirements are imposed:

a. Modularity. Computer programs shall be modular in design. Module identification shall be along functional lines with ease of maintenance being a prime consideration. To the maximum extent practical, data base information shall not be provided as in-line code. Rather, data shall be provided in a separate, non-executable module or file.

b. Structured programming. The principles of top-down, structured programming shall be used to the maximum extent practical. Each module or submodule of the computer program shall be designed with a single entry point and a single exit point.

c. Comments. Computer program listings shall contain comments that completely describe the functions being performed in each program module.

A3.3 Software Characteristics

The most common categories of software developed are embedded operational programs and ground support systems, particularly for system tests using ATE. The computer programs range from small (under 16K statements) to large (64K to 200K statements), the project size ranges from small to medium, and the development periods are relatively short (between 1 and 3 years). Emphasis is placed on the adequacy of documentation, with the following document items being representative:

System specification.
Computer program development specification (Part I).
Computer program product specification (Part II).
Computer program development plan (CPDP).
Configuration management plan.
Interface control document.
Operator's manual.
User's manual.
Computer program test plan and procedures.

The criticality of the computer programs developed at this site ranges from zero to two (see table 2.2-1 for explanation). Ground support and system test programs are considered criticality zero, while operational programs are either criticality one or two. Missile and weapon systems software and flight control systems software are considered more critical than telemetry, simulation, display, and scoring calculation programs. However, no distinction was evident in the level of requirements for criticality one or two software.

Only general information on the characteristics of the software was obtained for the three categories of systems developed at the site (threat, missile, and scoring systems). These are discussed in the following paragraphs.

A3.3.1 Threat Systems

The principal languages used in program implementation are FORTRAN-77 and some assembly language routines for special processes, such as input/output handling. The software for these systems has been typically programmed on minicomputers, such as the ECLIPSE, NOVA-3, VAX-11, HP 1000, ROLM, and PDP-11/LSI-11. The system functions they perform include ground systems, antenna control, network interfacing for mission data, servo/slave control, and target scenario simulation (used in electronic warfare air crew training). Representative functions that are implemented in software include the following:

Interface to keyboard, CRT, and disk.
Radar ranging and position calculation.
Servo positioning.
Message handling.
Radar systems monitoring.
Console control interface.
Radar systems simulation.
Tracking control.
Target and threat data interpretation.

A3.3.2 Missile Systems

The principal languages used for embedded operational computer programs are JOVIAL J73 and assembler. Because of the throughput performance requirements and limitations on available onboard memory inherent to missile systems, assembly language is commonly used to program the missile-resident computer programs. JOVIAL is used in non-missile-resident software and ATLAS is used for ground checkout programs. The onboard programs typically occupy 40K to 62K bytes of memory. The functions performed by the onboard programs include the following:

Navigation.
Autopilot.
Executive control.
Guidance.
Tracking and stabilization.
Fusing.

Downlink telemetry.
Electronic counter-countermeasures.
Built-in-test.

The ground systems developed for missile contracts perform the following functions:

Data link interface.
Command receiver.
Radar processing.
Ground test.

The ground test (ATE) software is developed and executed on minicomputers, such as the PDP-11; flight software is targeted for execution on a special-purpose 16-bit microprocessor, such as the General Missile Processor.

A3.3.3 Scoring Systems

Examples of scoring systems are the Digital Doppler Scoring System, Antenna Identification Scoring System, and Aerial Gunnery Target System. These systems typically are targeted for 8-bit or 16-bit microprocessors, such as the Intel 8080 and 8086. They are often coded in assembly language, using the Intel Development System. The software functions performed encompass the following:

Graphics.
Trajectory calculation.
Performance and statistics calculation.
Trajectory calculation algorithms.
Analog-to-digital conversion.
Telemetry (discrete and continuous).
Front end formatting and filtering.

A4. CATEGORIZATION OF SOFTWARE FUNCTIONS

The list of software functions in this appendix is based on responses to surveys performec as part of the preparation of this handbook. The first draft of this list was reviewed by representatives of this Air Force mission. However, it is possible that the list is not complete, or that another individual from the same organization would have described or

categorized the software types differently. The number that follows each software function is the assigned software category. This software category is 1 of 18 standard categories defined in section 2.2, and it is used in path 1 to determine the applicable testing techniques.

1. Threat Systems—defensive and offensive radar, targeting/tracking systems, electronic countermeasures

SOFTWARE FUNCTIONS	CATEGORY
controls and displays	14
message handling	8
radar range and position calculation	5
radar systems monitoring	10
servo positioning	10
target/threat data interpretation	16
tracking control	3

2. Guided Weapon Systems—air-to-air, air-to-ground missiles, smart bombs

SOFTWARE FUNCTIONS	CATEGORY
autopilot	6
built-in-test (BIT)	9
downlink telemetry	16
electronic counter-counter measures	10
executive control	4
fusing	2
guidance	3
launcher sequencing	3
mission data preparation	12
navigation	5
tracking and stabilization	3

3. Scoring Systems—proximity detectors for projectiles; used in aerial gunnery training

SOFTWARE FUNCTIONS	CATEGORY
analog-to-digital conversion	13
front end formatting and filtering	13
graphics	14
performance statistics calculation	1
telemetry	13
trajectory calculation	5
trajectory calculation algorithms	5

APPENDIX B: AVIONICS

The software testing procedures described in this appendix are based on a representative site devoted largely to this mission. Therefore, the procedures covered in this appendix may not include all aspects of the avionics mission.

B1. AERONAUTICAL SYSTEMS DIVISION (ASD)

ASD is a developing agency for weapons systems equipment, including avionics, automatic test equipment, crew training devices, and flight control and reconnaisance/C3 systems. System and software development are typically contracted out; the development contractors tend to be medium to large size aerospace corporations, with substantial technical expertise in weapon systems development. The systems and embedded software are developed under well-defined contractual requirements and monitored by on-site representatives and frequent reviews of activity and documentation by ASD personnel. A wide diversity of software is developed by ASD, including numerous aircraft avionics and control systems, and communications systems software.

Development activities are controlled by Government standards and testing practices are fairly uniform, adhering to AF regulations and uniformly defined testing requirements, imposed by SOW.

B2. ASD MISSION

The primary mission of Aeronautical Systems Division (ASD) is to acquire aeronautical systems that meet the needs of Air Force users such as Strategic Air Command, Tactical Air Command, Air Training Command, and Air Force Logistics Command to provide maintenance and support systems. Virtually all systems and equipments are developed by contractors, who are also responsible for development (and sometimes maintenance) of the computer programs and computer data.

Since there is very little weapon system software developed by ASD personnel, the main activities of ASD software engineers and software managers are to (1) assist in preparing the computer resource elements of specifications and requests for proposals, (2) participate in evaluating contractor proposals during the source-selection process, (3) monitor

the progress and change activity during system development, and (4) assess the degree to which requirements are being satisfied. These software engineers and managers interact with other engineers and managers involved with the system and report their findings through appropriate ASD internal channels to program management for status, understanding and decisionmaking. In certain programs, Government personnel are assisted in their tasks by support contractors commonly called Systems Engineering and Technical Assistance and independent verification and validation (IV&V) contractors. IV&V contractors are usually focused on software issues, while SETA contractors have a broader scope with software as one of the elements within that scope.

B3. SOFTWARE DEVELOPMENT ENVIRONMENT

This appendix covers five major types of weapon system software developed at ASD, including the development and mission support software associated with each type. These five types are categorized for convenience as avionics; automatic test equipment; air crew training device (ATD); flight control; reconnaissance; and command, control, and communications (C^3). The software developed by the ASD contractors is highly varied from category to category. Furthermore, the software within each category is far from homogeneous in its nature.

Avionics Software at ASD (Overview). Avionics software usually encompasses the software aboard aircraft, airborne strategic missiles, and some air-to-ground missiles acquired by ASD. Aircraft avionics operational flight programs (OFP) are generally divided into two categories: (1) offensive avionics, which implements functions such as navigation; air data computation; weapons management; sensor data reduction and controls; stores management; target recognition and designation; cockpit controls and displays terrain avoidance; terrain following; computer executive functions; and communications and (2) defensive avionics, which focuses on electronic threat detection; threat discrimination; threat avoidance; a variety of jamming techniques, controls, displays; and computer executive functions. The specific set of avionics functions depends on the nature of the aircraft (air-to-air fighter, air-to-ground fighter, multirole fighter, fighter bomber, strategic bomber, cargo, tanker, reconnaissance, or trainer) and the specific requirements for that aircraft.

Onboard automated built-in-test functions or central integrated test systems are used to determine hardware and system failures, to notify air crews for assessment of the effect on mission performance, and to notify ground crews for maintenance actions. These software functions may or may not be considered part of the avionics, depending on individual perspectives within ASD.

Detailed information follows regarding avionics software.

a. Requirements. The software functional and performance requirements are generally derived by contractors from avionics system requirements of the same type. In addition, the Air Force, may specify certain software design requirements that may change over the years, depending on the advancement of technologies.

   Generally, modern avionics software and firmware is distributed among special-purpose computers of the "minicomputer" class and among microprocessors. Communication among processors is usually according to the protocol defined for serial multiplex buses in MIL-STD-1553B. The OFP is a real-time program usually operating under an executive concept of rigorously scheduled function execution in the foreground activities for each mode of the mission. The scheduling timeframe for foreground is a part of the design of the software, based on how frequently the required data are updated to achieve mission performance. Less critical functions operate in the background and are usually scheduled on a time-available basis, with higher priority activities interrupting those of lower priority. This rigorous scheduling is imposed to simplify the design concept and to ensure repeatability during software and system test so that transient anomalies are minimized.

b. Language. Until the B-1 program and the F-16 program, avionics software was exclusively programmed in assembly language. With the susccessful use of the JOVIAL language on the above two progams, the Air Force has standardized on the JOVIAL J73 language (according to MIL-STD-1589B) for all avionics application (unless an approved waiver based on technical or cost issues is granted). Offensive avionics software using JOVIAL usually has about 80% of the object code generated from JOVIAL source code with the remainder in assembly code. Input/output functions (not supported by JOVIAL) are relegated to assembly code. With this 80/20 mix, the JOVIAL expansion of code and timing is estimated at 15% over that of well-done assembly code.

c. Size. The size of avionics OFP's varies considerably from application to application. For simple fighter aircraft, the OFP may be less than 3,000 or 4,000 instructions; whereas, for a strategic bomber the total OFP size may be 100 times larger. As the functional requirements increase, so do the sizes of the OFP and its data tables. The Air Force usually requires a margin on sizing, timing, and communications throughput to provide for future modification and growth of the avionics suite. These margins may be initially specified in the range of 15% to 50% of total capacity, but the tendency has been for software growth to use a significant amount of this margin before development is complete.

d. Development facilities. Typically, avionics software is developed on a general-purpose host (IBM 370, VAX 11/780) in JOVIAL and initially targeted for the host. After error-free compilation of units, some minor checkout takes place on the host. Units are then recompiled on the host and targeted for the flight computer. Some unit and module testing is done through an instruction-level simulation or interpretive computer simulation on the host, but this is usually minimal because of long running time and slow turnaround. A software development laboratory (SDL) is used for module and computer program component checkout and integration. This laboratory simulates (usually on Harris type or VAX computers) other system hardware elements and drives the software in real time on breadboard processors. Each processor in the distributed system is at first driven "standalone" to test the software in that one computer.

A more extensive facility, the System Integration Laboratory and Test Facility (SILTF) is used by the software development team to integrate the various computer elements so that the distributed system may be exercised with as much real equipment (rather than simulated equipment) as economical. After this phase, the software is handed over to a separate test team (within the same company), which conducts tests on the SILTF according to rigorous test plans and procedures developed by the contractor and reviewed by the Air Force. Digital and graphic data are recorded to verify correct functional performance against the verification cross-reference matrix contained in section 4 of the CPCI development specifications. Software problem reports or discrepancy reports are written by the software development team, and corrections prepared and periodically incorporated through contractor configuration control procedures. Retest is accomplished after corrections are implemented.

Test Equipment Software (Overview). There are three generic types of automatic test equipment that are procured by ASD for aircraft: flightline test equipment, intermediate shop test equipment, and depot test equipment. The purpose of the flightline ATE is to check out aircraft systems to isolate faulty black boxes, or line replaceable units (LRU). The intermediate shop test equipment designed for combat bases uses ATE to isolate faults in LRU's to specific electronic cards, or shop replaceable units (SRU), which are then replaced. SRU's are either discarded or sent to one of five U.S. depots where the third type of ATE isolates the faulty components on the cards, which are then repaired and returned to inventory.

Detailed information follows regarding test equipment software.

a. Software categories. There are usually four categories of software that are part of the ATE software: (1) the operating system, which is usually provided by the vendor of the computer (often a commercially available machine); (2) the support software, which includes compilers or interpreters, assemblers, linkers and loader (often commercially available); (3) control software, which controls the various electronic devices (signal generators and the like) that are part of the tester; and (4) unit-under-test (UUT) software, which activates in sequence the control software and measures the appropriate response of the UUT for that test condition. The UUT software also identifies the faulty elements of the UUT. The two most modern ATE developments are those for the F-16 aircraft and the Modular Automatic Test Equipment (MATE) program.

b. Language. The ATE software (for the first three categories above) is usually written in assembly language and FORTRAN, with the UUT software usually written in some version of ATLAS. The current Air Force standards are IEEE Standard C/ATLAS 716-1982 and 717-1982. The MATE program is an effort to standardize on ATE interfaces for future equipments. MATE is examining the use of JOVIAL J73 rather than FORTRAN for future ATE software and is advocating the use of the above IEEE standards for ATLAS.

c. Development process. The requirements process usually starts with a Test Requirements Document (TRD) for the equipments to be tested by ATE. The TRD's are usually prepared by the designers or manufacturers of the units. From the TRD's and previous experience, a weapon system contractor or an ATE contractor derives

or selects the test station requirements, which include the operating system and control software requirements. From the TRD's and detailed documentation on the UUT's and test stations, the specifications for UUT software are derived. In more recent systems, these documents have been reviewed by IV&V contractors, which the SPO's have indicated to be beneficial and cost-effective.

d. Testing. For economic reasons, the tests to be implemented in ATE software can never be totally exhaustive. There are many failure modes possible to a UUT, either singly or in combination; consequently, those that constitute a high percentage (90% to 95%) of all failures are implemented in the UUT software.

It is not economical to test every fault isolation option in UUT software on actual test hardware. Since the fault itself must be inserted into the UUT to conduct the test (this may be difficult to do without inserting multiple faults) and since the number of test units available may be limited, UUT software testing usually is slow and expensive. Usually for tests witnessed by the Air Force, 50 to 100 different tests are run on the software against a single, actual UUT. Remaining errors, problems, or needed test programs are resolved during the software maintenance activity, with some economic justification.

e. Special requirements. Fault tolerance is seldom a requirement for ATE software. If there is a hardware or software fault in the tester or the UUT, the philosophy has been that the fault should be repaired rather than provide software to work around that fault. Self-test is usually provided in the test station so that test station hardware failures may be isolated and repaired quickly to bring the test station back on line.

f. Development environment. Modern systems normally compile, interpret, and assemble on the test computer. An exception to this is the F-15 intermediate shop test equipment system, which compiles the UUT software on an IBM 360. This approach is now deemed to be less efficient. Much of the debugging is done on the tester itself with a real hardware UUT in place, turn-around through a separate host takes too long, and the minicomputer in the test set has the capacity to do the hosting job.

Neither environmental simulators nor simulations of the UUT are used. The first is not needed and the second approach is not considered effective. Writing and debugging the simulation of the UUT is considered more expensive than the present methods that use the actual UUT hardware.

Simulator Software (Overview). ASD acquires a variety of automated crew training devices, ranging from simple part-task trainers, which provide a training element that may be as short as 8 to 10 minutes in duration, through full weapon system trainers, which may simulate an 8- to 10-hour mission for an entire strategic bomber crew, including pilot, copilot, navigator, flight engineer, and electronics warfare officer. For pilot training, these air crew training devices present all of the cockpit instruments and displays, pilot controls, window and heads-up displays, motion effects, aural cues and effects, realistic aerodynamic response for the simulated aircraft, engine responses, vibration, and avionics equipment behavior, including sensor behavior and weapon release.

a. Virtually all of the functions are simulated in software on one or multiple commercially available 32-bit minicomputers. In a recent system (F-16), a copy of the aircraft avionics computer with its flight software has been included in the simulator itself, rather than simulating the flight programs on a general-purpose computer. Commercially available operating system software, peripheral devices (tapes, disks, printers, CRT's, etc.) and their control software are generally used. Applications programs that simulate the aircraft function and perform much of the instructor station operation are specified to be written in FORTRAN.

b. Software characteristics. The fundamental control philosophy for simulator software design is a prescheduled, synchronous timeframe for those highly cyclic aircraft activities with other lower priority activities running in the background on an interruptible basis. Modularity (one function per module), top-down design, and separation of data from program instructions are usually requirements on the software effort. The software generally checks the ranges and validity of instructor-supplied parameters and provides fault data for maintenance purposes.

c. Specifications. Usually the entire ATD is a single configuration item of which the software is an element. The system specification for the ATD indicates the functions to be represented in the system, the general level of fidelity, the required margins for computational speed, bus throughput, directly addressable memory, and bulk memory (usually disk).

d. Testing. The fundamental adequacy of the software and simulator performance is judged through formal tests in which experienced pilots "fly" the simulator and assess its "feel" compared to the real aircraft. Other elements of the system, such as the instructor station capabilities, are verified by objective tests.

Flight Controls (Overview). Digital flight controls and digital engine controls represent relatively new areas of computer application at ASD. Both involve the primary issues of high performance and flight safety. Whereas the computer resource activity for the three applications previously discussed have had at least 10 to 15 years of history and evolution at ASD, these control applications are relatively new and are not yet implemented in an aircraft scheduled for production.

a. Language. Development of flight control software has to date been in assembly language but will no doubt be done in HOL in the future as mature compilers become available. The general development and test procedures are similar to those for avionics software.

b. Software characteristics. The control law implementations for multiaxis stability and aircraft control are highly algorithmic in nature with different algorithms and different control gains for different flight modes or regimes. The software is written against prescheduled time increments so that periodic data updates and control computations are completed at a cyclic frequency to maintain an adequate margin for stability and control. Less important functions are scheduled in the background, some of which may be triggered on an event rather than on a cyclic basis

The preparation of flight control software requires a thorough understanding of the hardware implementation, both in its failed states and its unfailed states, as well as an understanding of control theory. The testing of this software is complicated by the requirement to test the system both in its nomimal state and in its large number of failure combinations.

c. Testing. ASD/ENF is currently determining what will be necessary in the software requirements and test area for safety qualification. The character of flight control software can be generally ascertained by a review of the advanced fighter technology integration (AFTI) programs being pursued by the Flight Dynamics Laboratory in the Air Force Wright Aeronautical Laboratories.

d. Special requirements. In general, the sensor and computational hardware will be triple or quadruple redundant so that failure of one or two processors would not jeopardize flight safety. Sensor data will be cross-strapped among the processors so that each can operate on the full set of sensor data with identical software. Comparison of input data from similar sensors will be used to isolate failed sensor strings. Comparison of output data will be used to identify failed processor elements. The fault-tolerance requirement (hardware fault detection and isolation) is a key requirement and adds considerable complexity, particularly if battle damage causes aerodynamic and control surface changes.

Overview of Reconnaissance and C^3. ASD is responsible for the acquisition of ground systems that receive (from aircraft sensors) data regarding the ground threat environment. These data, such as radar digital maps or ground electronic emissions information, are processed to determine the nature and location of various threats. Upon threat identification and location (during a real battle), the ground computational system, in conjunction with human controllers, may (1) plan an action against some of the threats, (2) allocate weapon and aircraft resources, and (3) control the flightpath of the aircraft and/or its weapons to the vicinity of the selected targets.

a. Hardware. These systems may be implemented in commercially available or militarized versions of commercial computers. These are usually of the mini or super minicomputer class with special-purpose, high throughput processors for signal processing and distributed, tightly coupled parallel processors for the remainder of the processing.

b. Software. FORTRAN and assembly are usually the languages used for the application software that is structured and modular. The development is usually on the minicomputers used for the project, and the testing procedures parallel those used in the SILTF.

Future flight control software will interact with the avionics software (1) to achieve automated delivery of weapons and (2) to use avionics sensors. A key requirement on this type of software will be that errors in the avionics system shall not propagate into the flight control software. This may be a difficult requirement to validate.

B4. CATEGORIZATION OF SOFTWARE FUNCTIONS

The list of software functions in this appendix is based on responses to surveys performed as part of the preparation of this handbook. The first draft of this list was reviewed by representatives of this Air Force mission. However, it is possible that the list is not complete, or that another individual from the same organization would have described or categorized the software types differently.

The number that follows each software function is the assigned software category. This software category is 1 of 18 standard categories defined in section 2.2 and is used in path 1 to determine the applicable testing techniques.

A. Airborne Systems

1. Avionics Systems

a. Mission Avionics

SOFTWARE FUNCTIONS	CATEGORY
aerial delivery	3
automatic approach/landing	4
communication	10
control/display processing	14
data bus control	2
navigation/guidance	5
real time executive	2
self-test	9
sensor control	3
sensor data reduction	10
sensor test/credibility	10
terrain following/avoidance	2,16

b. Offensive Avionics

SOFTWARE FUNCTIONS	CATEGORY
stores management	1
target recognition/acquisition	16
weapons control	3

c. Defensive Avionics

SOFTWARE FUNCTIONS	CATEGORY
jamming	10
threat avoidance	2
threat detection	10
threat discrimination	16

2. Flight Critical Systems

a. Flight Control

SOFTWARE FUNCTIONS	CATEGORY
digital flight control	3
sensor data processing	10

b. Fuel Management 2

c. Engine Control

SOFTWARE FUNCTIONS	CATEGORY
digital engine control	3
engine cycle data acquisition	13
fault detection and accommodation	2,9

B. Ground Systems

1. Air Crew Training Devices

SOFTWARE FUNCTIONS	CATEGORY
aural system	3
computation system (executive, support, maintenance software)	1,4
digital radar land mass system	3
electronic warfare system	3
electro-optical viewing system	3
gravity seat systems	3
instructor/operator station	14
motion systems	3
student station	2
visual systems	14

2. Automatic Test Equipment

SOFTWARE FUNCTIONS	CATEGORY
control software	2
system software (compilers, support s/w)	17
Unit Under Test (UUT) software	9

APPENDIX C: COMMAND, CONTROL, AND COMMUNICATIONS

The software testing procedures described in this appendix are based on a representative site devoted largely to this mission. Therefore, the procedures covered in this appendix may not include all aspects of the command and control mission.

C1. STRATEGIC AIR COMMAND

The Strategic Air Command (SAC) has a diversity of missions to support, such as command and control, war planning, intelligence support, and strategic weapons support. It also develops a wide diversity of unrelated systems for these missions. For strategic weaponry, SAC is a user agency, while for the other areas it is both a developer and user. War planning and intelligence systems are developed and maintained almost exclusively by Air Force personnel, while often the development of information and management systems are primarily conducted by contractors with the maintenance shared by Air Force and contractor personnel. The software developed for the warning functions ranges from highly critical to noncritical. Software development practices for contractors are controlled by the SOW; internal maintenance is conducted in accordance with SAC regulations. SAC computation systems tend to be data base and data processing intensive, such as in the intelligence and war planning areas. The warning area includes real-time control functions, and the command centers use C^3 technology software. SAC-conducted software testing practices and methods are standardized by SAC regulations; however, there exists variability in their application, corresponding to the differences in the software categories, criticality, and functional organizational practices.

C2. SAC MISSIONS

The missions performed by SAC include command and control, war planning, warning and intelligence support. The general functions performed within these missions are as follows.

Automated command control:

a. Collection of status-of-forces information on a near-real-term basis, using generalized information on a near-real-time basis and a generalized software system called the Force Management Information System.

b. All geographically dispersed SAC subordinate units are linked to headquarters computers via a Data Transmission Subsystem.

c. Command Post wall screen and printer displays provide data to the Commander-in-Chief Strategic Air Command (CINCSAC) and the Battle Staff concerning availability of resources for Single Integrated Operational Plan (SIOP) execution. Progress of force activity can be monitored as events materialize.

d. Support is provided to the Single Tanker Missions for worldwide Tactical Air Command (TAC) aircraft deployments.

e. Support is provided to the SAC aircraft contingency planning staff.

f. Software development support is provided for Numbered Air Force Control Systems.

g. Software development support is provided for Airborne Command Post Force Control Systems.

War planning:

a. Planning of intercontinental ballistic missiles (ICBM), aircraft, and cruise missile sorties against specified enemy targets is accomplished using intelligence estimates, weapons capabilities, and geological factors.

b. Computer simulations permit "flying" sorties to determine success probability.

c. Extensive use of interactive graphics permit SAC and JSTPS planners to visualize SIOP development.

d. Production of flight plan cassettes for unmanned cruise missiles.

e. Gaming techniques provide information on methods to improve the plan by pitting the SIOP against the probable enemy plan.

Warning:

a. Computers embedded in various missile warning field sensors enable the detection and/or tracking of hostile missile launches.

b. Near-real-time displays on several display devices notify Command Post personnel of endangered SAC resources and provide information needed for decisions of force posturing, including launch for survival of aircraft.

c. An automated countdown to impact and checklists of required actions greatly assist the decisionmaking process.

Intelligence support:

a. Online interactive analyst support is provided for collection management, photographic and electronic intelligence analysis and correlation, target development for the National Target Base and maintenance of offensive and defensive orders-of-battle on the SAC On-Line Analysis and Retrieval System (SOLARS).

b. Automated processing of electronic intelligence (ELINT) is accomplished to support the airborne reconnaissance program.

c. Development of processing systems provide for SAC evaluation of airborne reconnaissance collectors.

d. Use of graphic displays supports processing of scientific and technical data describing electronic emitter characteristics.

e. Map overlay plotting is used to support SIOP and ELINT production.

f. Automated support to reprogrammable airborne electronic warfare systems is provided.

g. Communications support and online analytical support for operational intelligence analysts are provided.

Management support:

a. The management information requirements of the HQ SAC staff are supported with 40 Air Force standard and 39 command-unique Management Information Data Systems.

b. Remote terminals in the Headquarters building permit online support.

c. Computer output microfiche (COM) capability is available, as well as the Honeywell Page Printing System.

d. Liaison is maintained with the Air Force Data Systems Design Center, Manpower and Personnel Center, Accounting and Finance Center, and other MAJCOM's.

C3. CATEGORIZATION OF SOFTWARE FUNCTIONS

The list of software functions in this appendix is based on responses to surveys performed as part of this handbook preparation. The first draft of this list was reviewed by representatives of this Air Force mission. However, it is possible that the list is not complete, or that another individual from the same organization would have described or categorized the software types differently.

The number that follows each software function is the assigned software category. This software category is 1 of 18 standard categories defined in section 2.2 and is used in path 1 to determine the applicable testing techniques.

SOFTWARE FUNCTIONS	CATEGORY
controls and displays	14
data base management	12
interactive interface	14
mapping/plotting (graphics)	14
mission data preparation	12,1
sensor data processing	10
simulation (non real-time)	11
simulation (real-time)	11
tracking	6

APPENDIX D: MISSILE/SPACE

The software testing procedures described in this appendix are based on a representative site devoted largely to this mission. Therefore, the procedures covered in this appendix may not include all aspects of the missile/space mission.

D1. SPACE DIVISION

The Space Division (SD) is a development agency for space-related systems, including satellites, launch vehicles, and ground control and communications systems. SD relies extensively on contractors to develop its systems and the embedded software, which also performs maintenance under follow-on contracts. Software development requirements are defined in detail in the SOW; and SD personnel, often coupled with technical consultant contractors, monitor all development activities at all levels intensively. Frequent reviews and technical direction are provided by this agency. A wide diversity of software categories is developed by SD, including software for communications, satellite control systems, prelaunch checkout and ground test systems, space vehicle avionics and control, and system simulations. This site employs IV&V contractors to a greater extent than any of the other sites surveyed. Software testing practices are established by Air Force regulation, defined by SOW and, as a result, tend to be relatively uniform among the development contractors. SD places great emphasis on the thoroughness, sufficiency, and formality of contractor testing practices.

D2. SD MISSION

The primary mission of SD is to acquire space-related systems, which include satellites, launch vehicles, and ground control and communications systems. Also, SD is responsible for managing and operating some elements of these acquired systems, such as the Satellite Control Facility and the Vandenberg Launch Facility. The new Space Command will impact these operating activities in a way that is yet to be determined.

SD relies on contractors to develop its systems, including the software within its systems. Development contractors for SD usually continue to maintain the software (if maintenance is required).

Because systems and system software are not developed by SD, the main activity of its personnel is to prepare RFP's, evaluate proposals, and conduct software management surveillance during the contract.

Technical assistance in the software area is provided by Aerospace Corporation, a non-profit systems engineering and technical direction contractor, providing technical consultation to the Air Force. In addition, particular major programs are usually technically assisted by IV&V contractors, who are selected competitively on a program-by-program basis.

D3. SOFTWARE DEVELOPMENT ENVIRONMENT

This section of the report will present an overview of four major types of SD software: ground control and communications systems, prelaunch checkout and launch systems, launch vehicle systems, and space vehicle systems. It will be evident that the software developed by SD contractors is highly varied in character from category to category. There is no technical detail concerning SD software that is true for all applications at this site. Following these overviews, the report will focus primarily on those specific applications encompassed in the survey.

SD relies on contractors using their own tools to develop and test software. Unlike the aeronautical systems, space systems rely on development contractors to continue to maintain the software through its life cycle; furthermore, for certain systems a high degree of contractor IV&V is provided. The software characteristics within an entire system and across systems vary substantially.

Ground Control and Communications (Overview). SD has acquired and is acquiring systems and software that (1) control the attitudes and functioning of unmanned satellites; (2) gather, reduce, record, and display data transmitted by satellites; and (3) communicate and control manned space activities. Some examples of these systems include the Satellite Control Facility (SCF) at Sunnyvale, the Global Positioning Satellite ground component, and the Consolidated Space Operations Center (CSOC) at Colorado Springs.

These systems have extensive amounts of software with major functions such as satellite detection (both enemy and friendly), orbit determination, displays of status to controllers, and satellite attitude correction and communication with one another and remote sites. In fact, if either the CSOC or the SCF become disabled, they can perform each others functions, as well as the Johnson Space Flight Center.

Most of the software is written in higher order language (JOVIAL and HAL-S with some FORTRAN for CSOC and JOVIAL for SCF) and operates in "near-real" time. CSOC software is maintained by Air Force personnel with substantive contractor support, while SCF primarily uses contractor maintenance. These systems are generically similar to SD's reconnaissance and C^3 systems but are larger in scope and more multipurpose.

Prelaunch Checkout and Launch Systems (Overview). A considerable investment in software resides in prelaunch checkout and ground launch systems that perform the booster and satellite ground checks before and during countdown and that peform the range safety function during launch. For new vehicles, the existing facilites are adapted, new software written, and additional factory checkout equipment moved to the launch site.

The software requires a detailed understanding of the hardware being checked and the system's function. These activities bear a simlarity to SD's automatic test equipment software, but on a more focused scale, since the checkout activity usually is confined to the contractor's facility and the launch site.

Launch Vehicle Systems (Overview). Software in launch vehicle systems maintains the stability of the vehicle during its flyout and takes inertial and other sensor information in order to follow a preplanned launch trajctory. The software design is based on a rigidly scheduled, cyclic sequence of events much like aircraft avionics software. This software is usually small in size, often fitting into a 16K word memory and is usually written in assembly.

The development and testing of this software parallels that described for SD's avionics software, except that it is simpler in function for the vehicles that launch unmanned payloads.

Space Vehicle Systems (Overview). Space vehicle systems may be classified as satellites that are manned, such as the Space Shuttle, and unmanned vehicles that are used for exoatmospheric transport, such as the inertial upper stage (IUS), and the Mini Vehicle used in the Antisatellite System.

For the most part, manned satellites do not use digital computers, aside from some recent systems that have small processors for attitude sensing and pointing and other station-keeping responsibilities. More use of digital computers in future satellites is expected, with emphasis for these computers on low power and fault-tolerant design.

The Space Shuttle has substantial onboard software, but this effort was primarily a NASA effort and beyond the scope of this study.

Exoatmospheric transport vehicles again are very similar to launch vehicles in their software matures, with the exception of the additional feature of engine control, more extensive maneuvering, and payload dispensing. Again, like aircraft avionics, simulation using a hot bench (SILTF-like facility) is performed during the design and testing to establish the real-time performance.

Usually, more extensive IV&V is performed on these systems than on SD systems. This IV&V usually includes independent testing on separate facilities, using separate tools by IV&V contractors.

D4. CATEGORIZATION OF SOFTWARE FUNCTIONS

The list of software functions in this appendix is based on responses to surveys performed as part of this handbook preparation. The first draft of this list was reviewed by representatives of this Air Force mission. However, it is possible that the list is not complete, or that another individual from the same organization would have described or categorized the software types differently.

The number that follows each software function is the assigned software category. This category is 1 of 18 standard categories defined in section 2.2 and is used in path 1 to determine the applicable testing techniques.

A. Equipment Checkout—pre-launch checkout, equipment self-test

SOFTWARE FUNCTIONS	CATEGORY
automatic test equipment (ATE)	9
built-in-test (BIT)	9
central integrated test systems (CITS)	9

B. Aerospace defense—threat detection and warning, threat evaluation

SOFTWARE FUNCTIONS	CATEGORY
automatic processing	13
data base management	12
filtering and smoothing	9
guidance and control	3
message processing	8
mission data preparation	12
mission planning	15
real-time control	2
real time executive	2
satellite impact prediction	5,7
satellite tracking	5
sensor processing	10
(sensor) tracking	5
simulation	11
situation notification	14
space information correlation	1
space situation analysis	15
task selection and displays	14

APPENDIX E: MISSION/FORCE MANAGEMENT

The software testing procedures described in this appendix are based on a representative site devoted largely to this mission. So the procedures covered in this appendix do not include all aspects of the mission/force management mission.

E1. TACTICAL AIR COMMAND

The Tactical Air Command (TAC) is the development and user agency for the major Air Force tactical planning system, Computer Assisted Air Force Management System (CAFMS). The CAFMS is a single-function, highly interrelated automated processing system. The major output product of CAFMS is the air tasking order report. CAFMS was developed by TAC personnel with some contractor assistance during the early requirements and design phases. Management, development, and maintenance of this system is well defined and uniquely adapted for its ongoing support. The system is currently operational, but undergoes continual enhancements and incorporation of new capabilities. The overall function of the CAFMS is quite critical, but few of its software components are considered to be more than moderately critical. The system does incorporate some automated fallback provisions in case of failure, but redundancy of systems function is not provided and reversion to manual operation is the ultimate fallback provision. Testing practices are well defined and are incorporated as an integral part of a version release management system developed by TAC specifically for CAFMS. Testing is applied uniformly to all software components undergoing development.

E2. TAC MISSION

The Tactical Air Command operates the Tactical Air Control Centers (TACC). The mission of TACC is to prepare, issue, and monitor the execution of coordinated orders for the employment of all forces available to the Air Force Component Commander. The TACC is the operations center of the Tactical Air Control System (TACS). The CAFMS was developed to augment the TACS with automated information processing, storage and display capabilities, and secure digital communications capabilities.

The primary mission area applicable to the TAC is Mission/Force Management. An

applicable secondary mission area is C^3, because the Mission/Force Management functions are integrated into and communicates through a communications network.

E3. SOFTWARE DEVELOPMENT ENVIRONMENT

This section provides a summary of CAFMS and its software development environment. TAC has no other tactical systems involving significant software development or maintenance that were applicable to the survey. All elements of CAFMS are developed, programmed, and controlled in the same manner and within the same organizational structure. The CAFMS is essentially a heterogeneous system in this respect. All code is implemented and tested according to the same standards and procedures. Therefore, the CAFMS was the only system surveyed for TAC. It is discussed as a single entity in this report, although in actuality it comprises a number of individual but interrelated computer programs. Each of these programs is developed by a uniform and disciplined management process.

Virtually all software effort on CAFMS is considered to be new development, as opposed to maintenance of existing program elements. This development effort involves augmenting the existing system with additional functions and integrating them into the overall design. It also involves major revisions to the system performance parameters, such as the data base contents, to enhance the system or to accommodate new functions. In this manner, the CAFMS is undergoing an evolutionary development process to meet current tactical planning demands and also to adapt it to changes in its operational environment. All changes are accomplished in a phased approach.

Development of CAFMS requirements was shared about equally between Air Force personnel and a supporting contractor. Design and development through initial installation were accomplished mainly by the Air Force, with only about 10% done under contract. Subsequent development and maintenance are entirely the responsibility of TAC. The system is currently undergoing initial operational test and evaluation.

Criticality factors for CAFMS include major mission impact, which probably is representative of Air Force mission/force management systems. The confidence level (see table B-1 in appendix B for explanation) that applies to software development and testing is level 2. Therefore, the development disciplines and level of software error detection are

comparable to many of the other major Air Force weapon systems, such as command and control and avionics systems.

CAFMS Overview. CAFMS is designed primarily to build, disseminate, and monitor the execution of the Air Tasking Order (ATO). There is also a requirement to build and generate a variety of status reports and periodic and end-of-day summaries. CAFMS reduces ATO preparation time. Since TACC is mobile, CAFMS must be capable of limited deployment. Therefore, there must be some ability to identify and change the names, locations, etc., of subelements in the data base. Also, CAFMS must be capable of processing classified information up to and including SECRET. CAFMS provides an automated assist to the manual system for some of its key functions. The main operating centers are the 9th Air Force, the 12th Air Force, and the USAF Tactical Air Warfare Center. CAFMS is intended to fulfill the following requirements.

a. Increase capacity and accuracy in the display of air situation and mission progress data.
b. Maintain status of bases and forces.
c. Significantly decrease the time required for preparation and dissemination of the ATO.
d. Significantly decrease the time used in routine and clerical tasks associated with mission planning.
e. Automatically generate and disseminate status and summary reports.
f. Provide terminals at the Control and Operations Centers, Air Support Operations Centers, Wing Operations Centers, and TACC.
g. Maintain status of communications, weather, munitions, etc.
h. Provide an offline AUTODIN interface from the TACC to any AUTODIN user, through the 407L System, TACS Communication System.

System Description. CAFMS has the following six major system functions:

a. Startup. The startup function initializes all other system functions during initial startup or during recovery. This initialization includes establishment of the system environment; for example, communications assignments for participating units, message alert routing, display access authorization, and system access authorization. The data base are initialized either to start clean or, if after a recovery, to start at the last saved position. Communications initialization facilitates hookup of all remote terminals and other communications links.

b. Console. At TACC, the console functions include the ability to build, update, and disseminate the ATO. It also includes the automatic building of mission schedule files to be used by current operations and report generation for the each day's activities. Console functions common to both the remote terminals and the TACC include log-in to gain access to the system, display printing capability, review of the ATO, update and delete capabilities for mission schedule and other files, input validation, and the display function itself.

c. Communications. CAFMS communications function provides the interface between the TACC and external elements not equipped with a remote terminal. This offline capability allows dissemination of messages (primarily ATO) through AUTODIN or the TACS internal teletypewriter (TTY) network.

d. System environment definition. This function provides the capability to maintain and change or update the system environment as necessary. This includes a capability to receive a printed listing of any specified system environmental data (e.g., message routing table).

e. Message processing. The message processing function provides the capability to prepare the JINTACCS ATO display formats for transmission to addressees not possessing a remote terminal. This conversion process or reformatting includes the insertion of header and trailer information. When the message has been formatted, it is stored in a message file and later output to the offline paper tape punch.

f. Shutdown. The shutdown function provides the capability for either an orderly termination of all computer system functions or, if necessary, an emergency termination. An orderly shutdown includes notification to all consoles and remote terminals that shutdown has started. All messages queued to the paper tape punch are completed. The system environment and necessary data base information are saved, as well as any recording information being generated. In accordance with appropriate security directives, memory and disk are overwritten. In the case of an emergency shutdown, only the memory and disk overwrite function are accomplished.

System Data Characteristics. For in-garrison operations, external data inputs are received by voice communications to the TACC. These data are manually entered into the system through local consoles. In deployed operations, inputs are provided through the remote terminals and/or voice communications. Functional user data inputs are as follows:

- Aircraft/Aircrew Status.
- Munitions Status.
- Weather Status.
- Unit/Base Status.
- Air/Ground Situations.
- Communications link Status.

The data outputs provided by CAFMS are the ATO message, Mission Schedule displays, and Status/Report displays. The following are the displays available in CAFMS.

- Air Tasking Order
- Mission Schedule Displays
  - Fighter/FAC/Support/Other
  - Reconnaissance
- Status/Report Displays
  - Unit/Base Status
  - Aircraft/Aircrew Status
  - Munitions Status
  - Weather Status
  - Aircraft Losses
  - Unit Air Sortie Recap
  - Mission Air Sortie Recap
  - Communication Circuits Status
- Strike Packages

Standards and Documentation. The major regulations applicable to CAFMS software development are the AFR 300-series and AFR 800-14, and DOD 7935.1-S, Automated Data Systems Documentation Standard. Applicable computer program documentation include the following items.

- System specification.
- Computer program design specification.

- Configuration management plan.
- Data base specification.
- Operator's manual.
- User's manual.
- Functional description.
- Development test plan (one per module).

Programming standards and conventions identified for CAFMS provide coverage for top-down structured development (analysis, design, and process), coding standards and testing requirements (module, subsystem, and system testing).

E4. CATEGORIZATION OF SOFTWARE FUNCTIONS

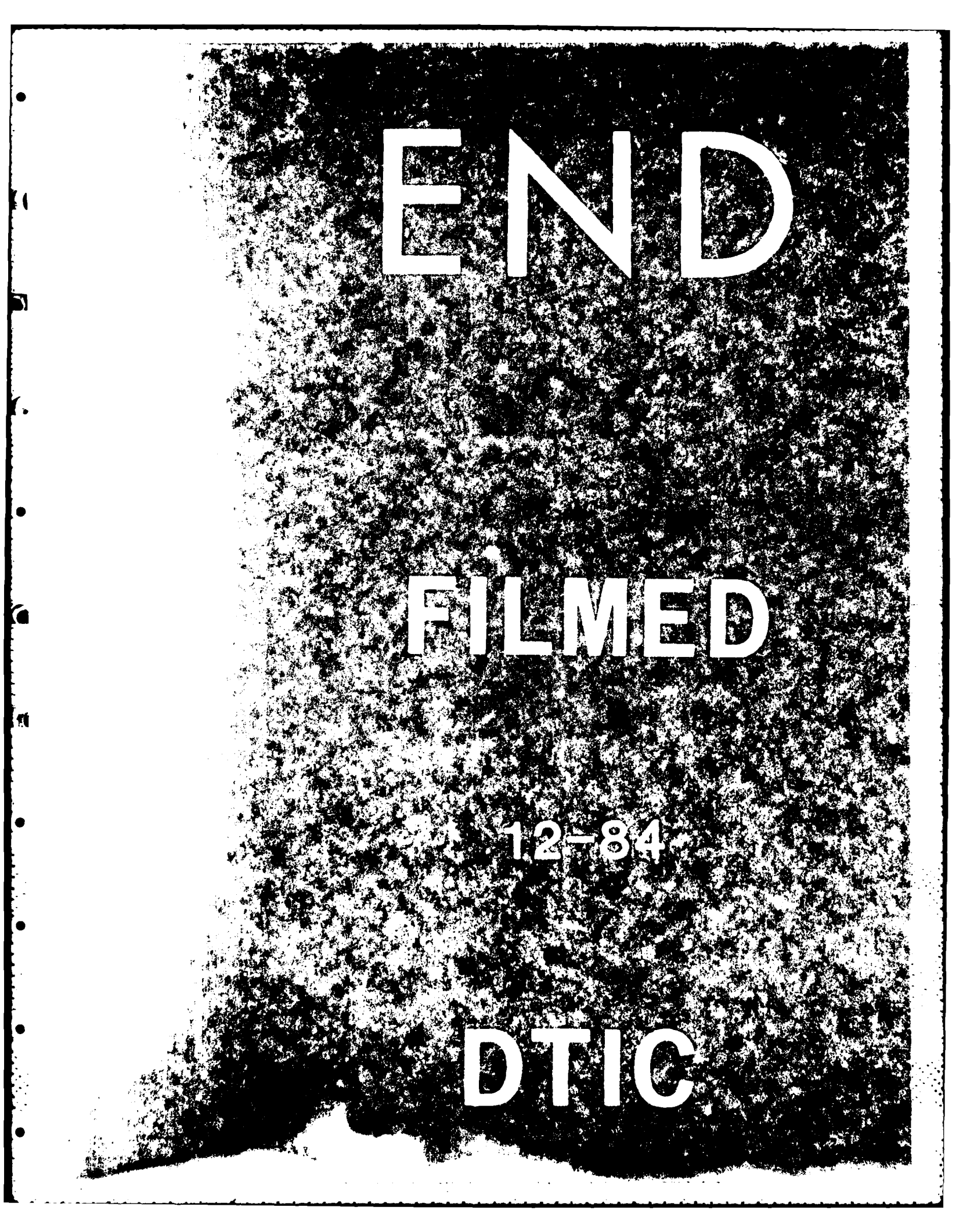
The list of software functions in this appendix is based on responses to surveys performed as part of the preparation of this handbook. The first draft of this list was reviewed by representatives of this Air Force mission. However, it is possible that the list is not complete, or that another individual from the same organization would have described or categorized the software types differently.

The number that follows each software function is the software category into which this function has been assigned. This software category is one of 18 standard categories defined in section 2.2 and is used in path 1 to determine the applicable testing techniques.

SOFTWARE FUNCTIONS	CATEGORY
communication	10
controls and displays	14
data base management	12
mapping	14
message processing	8
secure data processing	12, 8, 4
war planning	15

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


END
FILMED
12-84
DTIC